

Feedback Linux Scheduling and a Simulation Tool for Wireless Control

Martin Ohlin

Department of Automatic Control
Lund University
Lund, June 2006

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

ISSN 0280-5316
ISRN LUTFD2/TFRT--3240--SE

© 2006 by Martin Ohlin. All rights reserved.
Printed in Sweden,
Lund University, Lund 2006

Abstract

Computing systems are becoming more and more complex and powerful. It is nowadays not uncommon to run several server applications on the same physical platform. This gives rise to a need for resource reservation techniques, so that administrators may prioritize some tasks, or customers, over others. This thesis gives an introduction to the Linux kernel 2.6 task scheduler, and scheduling related operating system concepts such as priority, nice value, interactivity and task states. The thesis also presents an implementation of a scheduling mechanism, that in a non-intrusive way introduces per task CPU bandwidth reservations in the Linux operating system.

The MATLAB/Simulink-based simulator TrueTime is given a short introduction, and the wireless capabilities of the tool are described in more detail. TrueTime is a tool for co-simulation of real-time tasks, network communication, and continuous-time plant dynamics. The modeling of the common medium access control (MAC) layers of IEEE 802.11 and IEEE 802.15.4 is described, along with the radio model used. TrueTime's capabilities to simulate local clocks with drift, Dynamic Voltage Scaling, and battery powered devices are also presented.

Acknowledgments

There is a large number of people who have helped me putting this thesis together. Some did it knowingly, others by just being there. People who know me probably also know that out of them all, I value my wife Mari-Kristin and my son Rasmus the most. May they always have the highest priority in my world.

I would like to thank all the people at the department for the friendly and helpful atmosphere you provide. I would especially like to thank my supervisor Karl-Erik Årzén for his many ideas and also his support for mine throughout this work; Anton Cervin for being my co-supervisor, our discussions about real-time and TrueTime, and of course for his creation of the first EU script. Oskar Nilsson has been the perfect room mate since the day I started at the department and deserves a special thanks. Our discussions have ranged from the highly relevant to the totally irrelevant, but have always been fun. Without him, I would probably not be the proud owner of an mp3 player converted to a heart-rate monitor. Peter Alriksson, the guy next door, deserves a thank you for all the times he has interrupted our room's work-loaded atmosphere with a pleasant discussion about his newest idea or a raving about something else. Anders Blomdell and Leif Andersson are great persons to look up to in many ways and deserve a lot of credit for keeping our computer environment in such a mint condition. They seem to have an endless knowledge of computer-related stuff, but also succeed in many other areas. Together with Rolf Braun, they are a breath of fresh air in an otherwise rather theoretic department. To all of the people not mentioned – You are not forgotten.

Finally, I would like to thank my family again for all their encouragement and support in everything I do.



Martin

Contents

| | |
|--|----|
| 1. Introduction | 9 |
| 1.1 Motivation | 9 |
| 1.2 Outline and Related Publications | 10 |
| 2. Controlling Linux in a Nice Way | 12 |
| 2.1 Introduction | 12 |
| 2.2 Objective of Control | 14 |
| 2.3 Linux Scheduling Overview | 15 |
| 2.4 Policy, Priority and Nice Value | 15 |
| 2.5 Interactive or not? | 17 |
| 2.6 Scheduling of Tasks | 21 |
| 2.7 Model of the System | 24 |
| 2.8 Control Structure | 28 |
| 2.9 Controller Implementation | 29 |
| 2.10 Usage | 30 |
| 2.11 Experiments | 32 |
| 2.12 Taking the State Into Account | 36 |
| 2.13 Related Work | 40 |
| 2.14 Conclusions | 41 |
| 3. A Simulation Tool for Wireless Control | 42 |
| 3.1 Introduction | 42 |
| 3.2 Wireless Networks | 43 |
| 3.3 The TrueTime Simulator | 45 |
| 3.4 The TrueTime Wireless Network | 48 |
| 3.5 Implementation Details | 55 |

Contents

| | | |
|-----------|---|-----------|
| 3.6 | Other New Simulation Features | 59 |
| 3.7 | Simulation Case Studies | 62 |
| 3.8 | Related Work | 68 |
| 3.9 | Conclusions | 69 |
| 4. | Future Work | 71 |
| 4.1 | The Nice Controller | 71 |
| 4.2 | The TrueTime Simulator | 72 |
| 5. | Bibliography | 73 |

1

Introduction

1.1 Motivation

This thesis consists of two separate parts. They will each be given a short introduction here and a more detailed one in the corresponding chapter. The first part, found in Chapter 2, presents a method to control resource usage in a Linux/UNIX system. Resource in this specific case is CPU bandwidth, but can also be I/O, memory etc. In many cases it is not necessary to control resource usage. If there, for example, is plenty of resources available, then the control introduces an unnecessary overhead. But if the resources are scarce, or if there is a lot of competition for them, then it might be advantageous and in some cases even needed to control the access to them. A key example of where resources are very sparse is in small wireless devices. In our daily life we get in contact with a large number of these wireless devices – in many cases more than we would really like to. During the last years, the number of devices has nearly exploded and they are now truly ubiquitous. A lot of these devices function more or less on their own and do not need or want any interaction from other devices in the area. Others however interact heavily with their environment. One example of the latter is sensor/actuator networks. When developing a large network of cooperating nodes in that way, it is very useful to have a simulation tool that captures the relevant aspects of the setup. One such tool is TrueTime, which is introduced in more detail in Chapter 3.

1.2 Outline and Related Publications

This section contains the outline of the rest of the thesis, together with references to related publications.

Chapter 2: Controlling Linux in a Nice Way

This chapter gives an overview of task scheduling in the Linux 2.6 kernel. It explains scheduling-related operating system concepts such as priority, nice value, interactivity and task states. An implementation of a CPU bandwidth controller is described in detail along with a number of experiments. The chapter ends with a summary of related work.

Publications

Andersson, Martin (2006) “Controlling Linux in a Nice Way.” In *Reglermöte 2006*. Stockholm, Sweden.

Contributions This represents work performed by the author alone.

Chapter 3: Simulation Tools For Wireless Control

This chapter introduces the wireless capabilities of the TrueTime simulation tool. The wireless radio model is described, together with an overview of the WLAN 802.11 and ZigBee 802.15.4 medium access control (MAC) layers. Simulation of local clocks with drift and battery powered devices are also described. The chapter ends with a number of examples and a summary of related work.

Publications

Andersson, Martin, D. Henriksson, A. Cervin, and K.-E. Årzén (2005): “Simulation of Wireless Networked Control Systems.” In *Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC 2005*. Seville, Spain.

Henriksson, Dan, A. Cervin, M. Andersson, and K.-E. Årzén (2006): “TrueTime: Simulation of Networked Computer Control Systems.” In *Proceedings of the 2nd IFAC Conference on Analysis and Design of Hybrid Systems*. Alghero, Italy.

Andersson, Martin, D. Henriksson, and A. Cervin (2005): “TrueTime 1.3—Reference Manual.” Department of Automatic Control, Lund Institute of Technology, Sweden.

Contributions This represents work performed in close collaboration with Dan Henriksson and Anton Cervin. Henriksson and Cervin developed the TrueTime kernel and network block, while the author developed the wireless network block. The author has also developed the battery block, local clocks and the dynamic voltage scaling.

Chapter 4: Future Work

The thesis ends with some suggestions for future work.

Funding

The work has been partly sponsored by the EU/IST FP6 integrated project RUNES.

2

Controlling Linux in a Nice Way

2.1 Introduction

Linux started as a hobby project in 1991. It was initially created by a young student at the University of Helsinki by the name Linus Torvalds. In 1994 he released version 1.0 of the Linux kernel. This kernel and its successors has since formed the base around which the various Linux operating systems are created. The code for the Linux kernel and most of the available applications is freely available to everyone. This freedom has made it possible for a large number of people and companies to contribute to and improve the code. Linux has even for many people become a synonym for free software. As the source code is free, a number of organizations have released their own versions of the operating system. Some of these organizations are Debian, Fedora and SuSE. Linux has for many years been a large competitor in server systems, such as web, ftp, mail and file servers. During the last years, the interest in Linux from commercial companies has increased dramatically. Nowadays, large enterprises such as IBM and Hewlett-Packard take active part in the Linux development, and also ship Linux as part of their large server and cluster systems. Linux is, however, not limited to server systems. The popular operating system can be found in many embedded devices such as set-top boxes, wireless access points and mo-

bile phones from, e.g., Motorola and Samsung. Linux is also gaining market in the desktop area, although maybe a little slower. The cities of Berlin and Munich, both in Germany, and Vienna in Austria have already decided to migrate their desktop PCs from Windows to Linux.

When two or more tasks run on the same computer, they share resources such as CPU bandwidth, and memory and network capacity. The exact decision of how the resources are split between the tasks is often left to the operating system. In most cases, this deference of command is a good thing because it is not normally known exactly how important the mail client is compared to the web browser. In some cases, however, it would be advantageous if there existed a mechanism to specify exactly how important different tasks (or groups of tasks) are compared to each other. Take for example the simple case where you run a large MATLAB simulation on your computer. If you are alone on the machine, you will get close to 100% of the CPU bandwidth. But you will only get about 50% of the CPU bandwidth, if another user logs on to the same machine and starts up his own MATLAB simulation. The same thing may happen when you decode a movie, but in this case the performance loss will probably be more noticeable because the human brain is very sensitive to jerky playback of sound and video. You probably do not think that this behaviour is fair; after all it is your computer and you should have more resources than the other person. So how can you change this type of behaviour?

The presented type of problem does not only show up in simple cases such as this one, but also on more complex computer systems such as web servers, where some of the clients may be more important than others. In the same way, it shows up in virtual hosting where several servers are run on the same machine. The rest of this chapter will present a mechanism aimed at solving this type of problem.

2.2 Objective of Control

The objective of control has been to achieve some sort of CPU bandwidth reservation. This concept makes it possible to reserve a fraction of the CPU to a specific task or a group of tasks. Linux does not distinguish between processes and threads. Instead they are both called tasks. Even POSIX types of threads created using the Native POSIX Thread Library (NPTL) are called tasks, and from the kernel perspective they are all the same schedulable entity. Therefore the method which will be presented can be used both for threads and processes.

The developed method has been implemented as an add-on to the Linux 2.6 kernel. A key factor in the implementation has been to make it non-intrusive and preserve the way that the original scheduler works. This gives the benefit that the new features can be used without breaking things that worked before. It is also important to note that large changes to such a critical part as the scheduler is unlikely to gain acceptance from the Linux community. The proposed method makes it possible to create a number of virtual CPUs and give them each, e.g., 20% of the physical CPU bandwidth during a certain period of time. The presented method uses the nice value as control signal and the execution time as measurement signal to make the scheduler give the controlled tasks their specified amount of CPU bandwidth.

It may be argued that the presented problem can be solved offline by specifying a static nice value for each task. This is of course absolutely true, if you happen to have a static system where everything is known beforehand. That is, you know exactly how many tasks that are present in your system, their exact execution-time demands, and do not allow tasks to enter or leave the system. These premises are not likely to show up in an ordinary Linux desktop or server system and therefore it is necessary to introduce a feedback loop to cope with the unknown. In an ordinary computer system there is a lot of dynamics. This is due to the fact that tasks can arrive and leave the system at any time. They can also change their state and in that way consume more or less execution time. In Symmetric Multi-Processing (SMP) or Symmetric Multi-Threading (SMT) systems such as Intel's Hyper-Threading systems, tasks will also jump between processors in a (from the user's perspective) more or less random pattern. This causes the execution environment to change rapidly and therefore an ability to adapt to different situations is necessary.

2.3 Linux Scheduling Overview

The rest of this chapter covers uniprocessor scheduling in the Linux 2.6(.12) kernel. In the cases where the scheduler depends on parameters whose values differ between the supported architectures, the values defined for i386 have been used. The Linux 2.6 scheduler can be seen as divided into two parts. The first lightweight part is executed at every timer interrupt and can be found in the function `scheduler_tick(void)` in the file `<kernel/sched.c>`. This function checks if the currently running task has exceeded its given time slice. If it has, preparations are made to make sure that the main part of the scheduler is called when returning from the timer interrupt. The main part of the scheduler is the function `schedule(void)` which can also be found in `<kernel/sched.c>`. This function is responsible for deciding which task will run next and also performs the context switch. The mentioned timer interrupt occurs with a frequency of HZ^1 , which is set to 1000 in the 2.6.12 kernel. The time between two consecutive timer interrupts is called a jiffy and is the smallest amount of time that the Linux kernel keeps track of and consequently corresponds to 1 ms.

2.4 Policy, Priority and Nice Value

There are three different scheduling policies available in Linux, one for normal tasks and two for soft real-time tasks. All of the policies are preemptive, i.e., if a process with higher priority gets ready to run it will preempt the currently running process. The policies are:

- **SCHED_FIFO: First In-First Out scheduling**

This is a policy for soft real-time tasks. Tasks in this group always have higher priority than tasks in the normal `SCHED_OTHER` group and will therefore preempt all other running tasks in the system. A task belonging to this group will stay at the head of its priority list until it either yields or is blocked by an I/O request, after which it will be put at the end of the list.

¹Kernels ranging from 2.6.13 and upwards have the `HZ` value as a config option, which defaults to 250. See <http://lkml.org/lkml/2005/7/8/259> for details and discussions.

- **SCHED_RR: Round Robin scheduling**

This is also a policy for soft real-time tasks, and very similar to the SCHED_FIFO policy. Tasks are, however, only allowed to run for a specific quantum of time before they are moved to the end of their respective priority lists. A task that has been preempted by a higher priority task is upon resumption allowed to run for its remaining part of the quantum. Apart from the mentioned differences, the two policies are the same.

- **SCHED_OTHER: Default Linux time-sharing scheduling**

This is the normal time-sharing scheduling policy that is used for all tasks that do not have very special timing demands.

Valid priority values in the Linux scheduler are in the range [0, 139] with 0 being the highest priority and 139 being the lowest. Priorities in the range [0, 99] are reserved for soft real-time tasks using the SCHED_FIFO or SCHED_RR policy. From now on, it will be assumed that SCHED_OTHER is used if not explicitly stated otherwise.

In Linux, every task has a static priority that is often referred to as the nice value. The nice value lies in the range [-20, 19] with 0 or 5 being the default value, depending on if the task is run in the foreground or background. A large nice value means that the task is very nice to other tasks, i.e., its priority is low. It also means that the task will not compete so hard for CPU bandwidth, that is, it will cause minimum interference to other tasks. The opposite is true for low nice values. The nice value of a task can be set and changed using the nice² and renice user programs, respectively. The nice values [-20, 19] are mapped directly to the priorities [100, 139] as seen in Figure 2.1.

To make things a little more complicated, but also better, the scheduler does not use the static priority derived from the nice value directly. Instead it uses something called dynamic priority, which in essence gives bonuses to interactive tasks and penalties to non-interactive tasks. The maximum bonus is -5 and the maximum penalty is 5. This gives an effective priority which is in the interval *static_prio* ± 5, so changes to the static priority (or nice value) are always respected.

²See `man nice` for more details

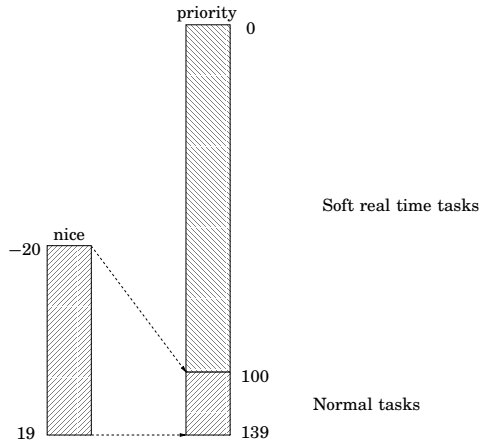


Figure 2.1 The picture shows how the nice value for normal tasks are mapped to priorities.

2.5 Interactive or not?

The Linux scheduler considers tasks to be either interactive or compute-bound. Interactive tasks get their priority slightly increased, depending on how interactive they are considered to be, while compute-bound tasks get it decreased in the same manner. This is a good thing, because you do not normally mind if your prime value calculations take some extra seconds to complete. But most people get very annoyed if it takes a noticeable time for the word processor to show the typed letters on the screen. The scheduler does not know beforehand which tasks are interactive, so it uses a heuristic method to decide which are. In essence, it measures the difference between how long time a task spends sleeping/waiting and how long time it spends running. The difference is saved in a variable called `sleep_avg` in the `task_struct`. Both the sleep-time and the run-time are truncated to the interval `[0, NS_MAX_SLEEP_AVG]` so that one long run-time or sleep-time does not affect the bonus system too much. The `sleep_avg` variable is also truncated to the same interval if needed. Truncation to an interval means that the truncated value is rounded up if it is lower than the interval, and rounded down if it is higher. After the truncation, the

`sleep_avg` variable is scaled to $[-5, 5]$ in the `CURRENT_BONUS(p)` macro, and this is what makes up the dynamic priority. As interactive tasks tend to wait for I/O most of the time, while non-interactive tasks do not, this is a fairly good measure. There are some special cases involved when updating the `sleep_avg` variable. First, there is a sort of low-pass behaviour that makes it easier for tasks to get interactive status than to lose it. Secondly, there are special cases for tasks that sleep very long or wait on I/O. There are also special cases that prohibit tasks that are very interactive to fork children which are also very interactive, and so on. This all means that tasks can be more or less interactive, but there are also cases when the scheduler wants a yes or no answer to this question. This is done using the `TASK_INTERACTIVE(p)` macro which looks at the tasks `nice` value and the current bonus. The mechanism used in this macro to decide if a task is interactive or not makes it hard for tasks with a large `nice` value (background tasks) to get interactive and vice versa. The `TASK_INTERACTIVE(p)` macro with its needed sub-macros can be viewed in Listing 2.1 and an attempt to visualize the result can be seen in Figure 2.3.

To be frank, the complete mechanism behind the interactivity estimation is very complex, but a starting point to understand the details would be to take a closer look at the function `effective_prio(task_t *p)`, the macros `TASK_INTERACTIVE(p)` and `CURRENT_BONUS(p)`, and also all updates of the variable `p->sleep_avg`, all found in `<kernel/sched.c>`.

A bug was found in the `TASK_INTERACTIVE` macro during the investigation of the scheduler code. The bug made the interactivity scale nonlinearly and was triggered when the `nice` value was negative, see Figure 2.2 and 2.3 for a comparison. The bug did not have any serious impact on the system, but was nonetheless regarded as a genuine bug. It is left as an exercise to the reader to find the bug in Listing 2.1. A report was sent to the Linux kernel mailing list [Andersson, 2006] presenting the problem along with a small patch which was promptly accepted and merged into the 2.6.17 version of the Linux kernel.

Listing 2.1 Code for deciding if a task is interactive or not, taken from `<include/linux/sched.h>` and `<kernel/sched.c>`.

```

#define MAX_USER_RT_PRIO      100
#define MAX_RT_PRIO          MAX_USER_RT_PRIO
#define MAX_PRIO              (MAX_RT_PRIO + 40)

#define PRIO_TO_NICE(prio)    ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)         PRIO_TO_NICE((p)->static_prio)

#define USER_PRIO(p)         ((p)-MAX_RT_PRIO)
#define MAX_USER_PRIO        (USER_PRIO(MAX_PRIO))

#define PRIO_BONUS_RATIO     25
#define MAX_BONUS            (MAX_USER_PRIO * PRIO_BONUS_RATIO / 100)
#define INTERACTIVE_DELTA    2

#define SCALE(v1,v1_max,v2_max) \
    (v1) * (v2_max) / (v1_max)

#define DELTA(p) \
    (SCALE(TASK_NICE(p), 40, MAX_BONUS) + INTERACTIVE_DELTA)

#define TASK_INTERACTIVE(p) \
    ((p)->prio <= (p)->static_prio - DELTA(p))

```

Chapter 2. Controlling Linux in a Nice Way

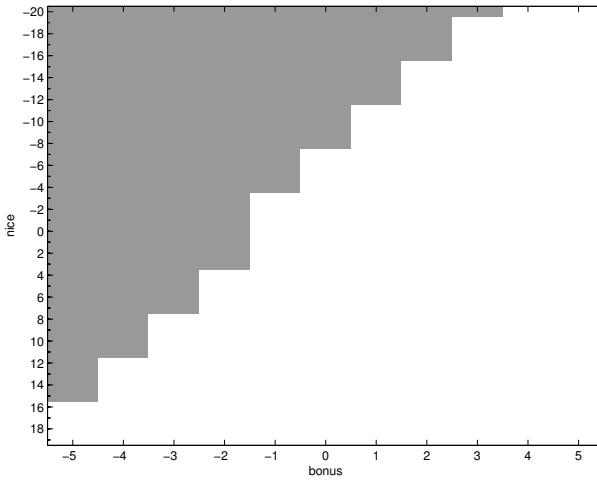


Figure 2.2 Picture showing how the nice value and current bonus give the interactive status of a task. Gray means that the task is interactive, white means that it is not. This is the version without the bug fix mentioned in Section 2.5.

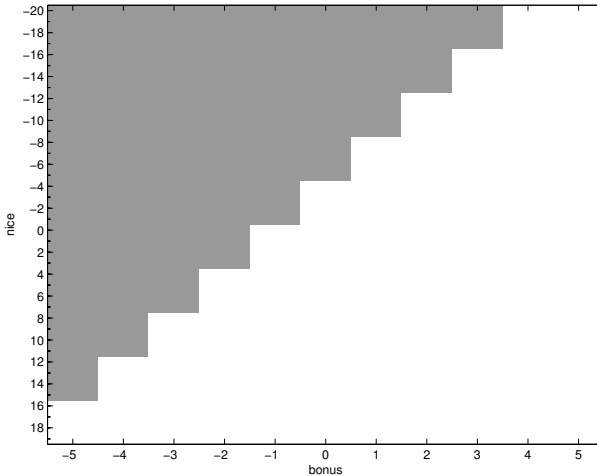


Figure 2.3 Picture showing how the nice value and current bonus give the interactive status of a task. Gray means that the task is interactive, white means that it is not. This is the version with the bug fix mentioned in Section 2.5.

2.6 Scheduling of Tasks

The scheduler has two priority queues, one for active tasks, and one for expired tasks. The queues are arrays of linked lists, one list for every priority. The scheduler always chooses the list with the highest priority in the active queue. The scheduler treats tasks differently depending on if they belong to the `SCHED_OTHER` policy or not. If we therefore assume that all tasks belong to the normal `SCHED_OTHER` policy, then the following is true. Every task in the active queue of the system gets to run a certain time before it is put in the expired queue. This time is called a `time_slice` and the actual size of it depends solely on the nice value given to the task and not on the effective priority. Nice values in the interval `[-20...0...19]` are mapped to time slice sizes in the interval `[800ms...100ms...5ms]` using the code in Listing 2.2. The size of the resulting time slices can be seen in Figure 2.4. Note that the resulting time slices do not scale linearly with the nice value.

A non-interactive task is moved to the expired queue when it has used up its given time slice, an interactive task on the other hand is reinserted into the active queue if there is no risk of starvation in the expired queue. When there are no more tasks left in the active queue, it is switched with the expired one. Whether there is starvation or not is decided in the `EXPIRED_STARVING` macro. The expired queue is considered to be starving if the first expired task has had to wait for more than a fixed time multiplied with the number of tasks in the active queue. This makes the starvation limit load depending. It is also considered to be starvation if a task with lower nice value than the currently running task is in the expired queue.

Tasks with the same priority are treated differently depending on whether the scheduler considers them to be interactive or not. Non-interactive tasks are not interrupted by tasks with the same priority during their time slice. Interactive tasks on the other hand get their time slice split up into smaller pieces of size `TIMESLICE_GRANULARITY` and are put at the end of the active queue again and again until they have executed for their whole time slice. The result is that interactive tasks are scheduled more or less round-robin with tasks of the same priority, while non-interactive tasks run non-preemptively. The size of the parameter `TIMESLICE_GRANULARITY` depends on the bonus given to the task.

Chapter 2. Controlling Linux in a Nice Way

Listing 2.2 Code for calculation of a task's time slice, taken from <kernel/sched.c> and <include/linux/sched.h>.

```
#define MAX_USER_RT_PRIO      100
#define MAX_RT_PRIO          MAX_USER_RT_PRIO
#define MAX_PRIO              (MAX_RT_PRIO + 40)

#define USER_PRIO(p)        ((p)-MAX_RT_PRIO)
#define MAX_USER_PRIO        (USER_PRIO(MAX_PRIO))

#define MIN_TIMESLICE        max(5 * HZ / 1000, 1)

#define NICE_TO_PRIO(nice)   (MAX_RT_PRIO + (nice) + 20)

#define DEF_TIMESLICE        (100 * HZ / 1000)

/*
 * task_timeslice() scales user-nice values [ -20 ... 0 ... 19 ]
 * to time slice values: [800ms ... 100ms ... 5ms]
 *
 * The higher a thread's priority, the bigger timeslices
 * it gets during one round of execution. But even the lowest
 * priority thread gets MIN_TIMESLICE worth of execution time.
 */

#define SCALE_PRIO(x, prio) \
    max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO/2), MIN_TIMESLICE)

static inline unsigned int task_timeslice(task_t *p)
{
    if (p->static_prio < NICE_TO_PRIO(0))
        return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
    else
        return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
}
```

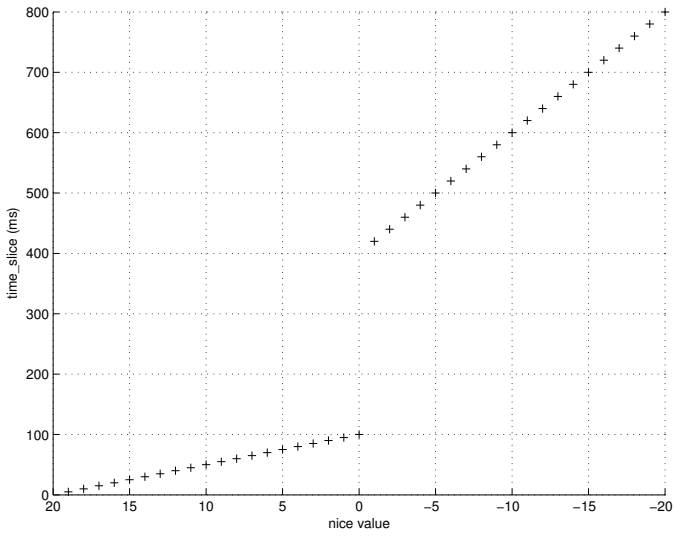


Figure 2.4 The size of `time_slice` as a function of the nice value.

2.7 Model of the System

Here we assume that a task is always willing to run, i.e., it is compute-bound. We can then summarize what we have learned in a few points and build a model of the system to predict its behaviour.

- A task's time slice depends solely on its nice value.
- Tasks are scheduled in order of priority.
- Tasks are moved from the active queue to the expired queue when they have executed their whole time slice.
- The active queue is swapped with the expired when empty.

According to this model, the fraction of execution time that a task gets during one round of execution can be calculated as:

$$fraction(i) = \frac{time_slice(i)}{\sum_{\forall j} time_slice(j)}$$

where i and j denote task indexes.

Example If, for example, tasks 1, 2 and 3 have nice value 0 and task 4 has nice value -1 , then task 4 will get:

$$fraction(i) = \frac{time_slice(i)}{\sum_{\forall j} time_slice(j)} = \frac{420}{100 + 100 + 100 + 420} \approx 58\%$$

Evaluation of the Model

To show that the presented model is accurate, a number of experiments have been performed. Theoretical values from the proposed model are compared to values received through measurements in a real-world system³. The experimental setup consists of four tasks running in an endless while-loop as seen in Listing 2.3. Three of the tasks have nice value five, while the nice value of the fourth task is varied in order to give it more or less CPU bandwidth compared to the other tasks. The expected result and the measured result can be seen in Table 2.1 in

³The authors normal desktop system with no special tweaks.

column two and three, respectively. A plot of the same values can be seen in Figure 2.5. Using lower nice values than -6 resulted in the system becoming to sluggish to do any good measurements, therefore these are left out. This sluggishness is probably due to the fact that tasks with that low nice values are given so high priorities that they conflict with the tasks interacting with the user. As can be seen, the results from the experiments follow the model very well.

Listing 2.3 Endless while loop used in the evaluation.

```
int main()
{
    int i = 1;
    int test;
    double test2;

    while(i){
        test2++;
        test++;
    }
}
```

Table 2.1 Comparison of the execution time model for Linux and measurements on a real world system.

| <i>nice value</i> | <i>Expected CPU%</i> | <i>Measured CPU%</i> |
|-------------------|----------------------|----------------------|
| -6 | 69.8 | 68.6 |
| -5 | 68.97 | 68.0 |
| -4 | 68.09 | 66.7 |
| -3 | 67.15 | 66.1 |
| -2 | 66.17 | 64.9 |
| -1 | 65.12 | 63.7 |
| 0 | 30.77 | 30.0 |
| 1 | 29.69 | 29.0 |
| 2 | 28.57 | 28.0 |
| 3 | 27.42 | 27.0 |
| 4 | 26.23 | 25.7 |
| 5 | 25 | 24.5 |
| 6 | 23.73 | 23.3 |
| 7 | 22.41 | 22.0 |
| 8 | 21.05 | 20.8 |
| 9 | 19.64 | 19.4 |
| 10 | 18.18 | 17.9 |
| 11 | 16.67 | 16.4 |
| 12 | 15.09 | 14.6 |
| 13 | 13.46 | 13.3 |
| 14 | 11.76 | 11.5 |
| 15 | 10.00 | 9.85 |
| 16 | 8.16 | 8.0 |
| 17 | 6.25 | 6.2 |
| 18 | 4.26 | 4.15 |
| 19 | 2.17 | 2.125 |

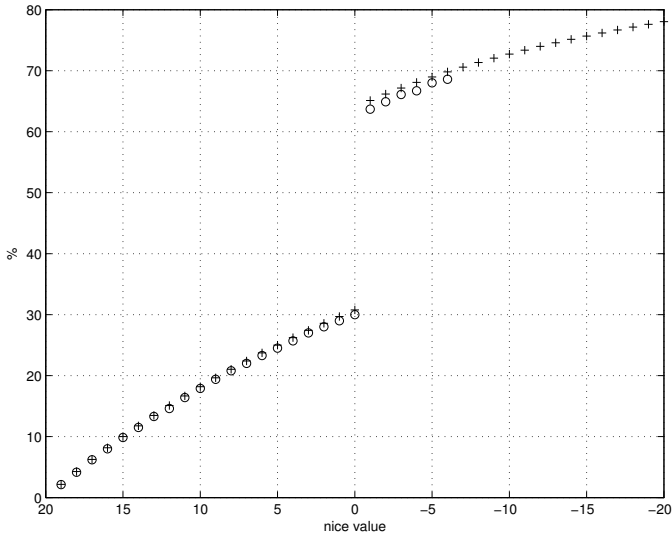


Figure 2.5 Comparison of the execution time model for Linux (+) and measurements on a real-world system (o).

2.8 Control Structure

Linux keeps track of all data belonging to a task in a structure called `task_struct`, defined in `<include/linux/sched.h>`. This structure contains, for example, the process identification number (`pid`), parent, children, siblings etc. The structure includes a lot of data relevant for control purposes such as a task's execution time, and the state of the task. The fields that we will use are `stime` and `utime`, which correspond to the execution time in kernel space and user space for the task, respectively. A task is said to execute in kernel space when it performs system calls, i.e., asks the operating system to perform some action on behalf of the task, such as opening or closing files or creating new tasks. All execution of code that is not done by the operating system is said to be done in user space. The granularity of the execution time measurement is one jiffy, which corresponds to 1 ms on the Intel architecture⁴. Both of the time counters are increased monotonically and are 32 bits long, which makes them wrap around after approximately 50 days if we assume that the task gets 100% of the CPU time.

The controller must run periodically, and should not be delayed for too long. This rules out the possibility to implement the controller in a user space task, as that would make its performance depend on other tasks in the system. The controller could, for instance, put itself into starvation if it increased the priority of another task too much. With this in mind, a Linux kernel module has been implemented that introduces the possibility to register tasks and give them a CPU bandwidth reference. Being in kernel space, the module is not vulnerable to the problem mentioned earlier, but on the other hand it makes the implementation much harder. The kernel module monitors the execution time given to registered tasks by periodically sample the mentioned `utime` and `stime` variables. The module also modifies the `nice` value of the task so that the bandwidth reference is met on average. This is done by creating a kernel timer, and register a function with it that is run every 20 ms. Every time the function is run, it goes through a list of registered tasks and saves the current execution time in a circular buffer. In this way it can output the execution time given to a task during, e.g., the last 20 ms. The kernel timer also executes a controller

⁴kernel 2.6.12

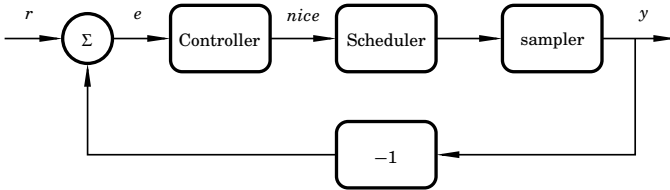


Figure 2.6 The setup used to control the task's execution time.

function that changes the `nice` value of the controlled task in order to try to get the fraction of time given to the task to correspond to the reference value. The structure of the setup with controller, scheduler and sampler can be seen in Figure 2.6.

The control signal, in this case the `nice` value, can only be changed in discrete steps. This makes it impossible to keep some references statically, but by changing very fast between two control signals, i.e., using Pulse Width Modulation (PWM), the reference can be kept on average. Another thing that one should be aware of is the nonlinearity in how the `nice` value is mapped to the `time_slice` as seen in Figure 2.4. This makes it much harder to follow references which forces the control signal to oscillate over the interval between 0 and -1 .

2.9 Controller Implementation

In this first implementation of the presented concept, the focus has been to get something that works in reality as a proof of concept. This has led to a design where a PI-controller with anti-windup has been implemented using fixed point arithmetics. In the future, more elaborate schemes could of course be used that take into account more details of how the scheduler works, as well as global knowledge of what other tasks are doing. By using a PI-controller in this way, the PWM behaviour described in Section 2.8, is achieved automatically. Some of the important parts of the controller implementation can be seen in Listing 2.4. The implemented PI-controller has $K = 0.01$ and $T_i = 52$. The integral part is implemented as a backward Euler approximation. The anti-windup works by limiting the integral state, if that state is

outside the control signal limits.

The K and T_i values have been chosen rather conservatively. This is due to the fact that even small changes to the nice value can lead to large changes in the achieved CPU-bandwidth. The effect of changes to the nice value also varies with the number of tasks in the system and their respective nice values in turn. The fact that there are unknown parameters in the system makes it good to have a conservatively tuned controller. Another thing that makes a conservatively tuned controller preferable, is the fact that measurements are taken over an interval. The system must therefore be given some time to react to the control signal before changing it again. Using a large value on T_i , also has the bonus effect that measurements are averaged; This removes the need to low-pass filter them.

When looking at the code in Listing 2.4, it is valuable to know that the external `set_user_nice()` function is $O(1)$ in complexity. This makes the overhead of changing the control signal very small. The low complexity of the function call is due to the fact that the scheduler uses a separate list for every priority. When a task's priority is changed, it is first removed from its current queue and then inserted at the end of the new one, i.e., no sorting is necessary.

In the implementation of the controller it has been assumed that right shift ($>>$), is done by shifting in the signed bit, i.e., arithmetic shift. This seems to be true on most architectures.

2.10 Usage

It is possible to communicate with the control structure described in Section 2.8 using the `/proc` interface. The `/proc` file system is a file system that does not exist on disk, it lives entirely in software. Every file is tied to a function in the kernel, which generates data on the fly when read. Files can both be written to and read from and gives therefore an easy way to communicate between user-space and kernel-space. The `/proc` file system is considered to be a standard way of getting data from kernel-space and is used heavily by many tools in Linux systems, e.g., `top` and `ps`. When the controller module is loaded, it creates a file named `/proc/fix/command` which can be used to send commands to and read some data from the module. To register a new

Listing 2.4 Implementation of the controller.

```

#define FIXPOINT                20
#define FLOAT2FIXED(a)         ((int)(a * (1 << FIXPOINT)))
#define K_float                0.01
#define K                      FLOAT2FIXED(K_float)
#define Ti                    52
#define I_faktor               ((int)((K_float / Ti) * (1 << FIXPOINT)))
#define umax                   (19 << FIXPOINT)
#define umin                   (-20 << FIXPOINT)

static void controller_fn(int pid, int error, int* I)
{
    struct task_struct *p;
    int u;

    u = K * error;
    u += *I;

    /* Backward Euler approximation of the integral part */
    *I = *I + PERIOD * I_faktor * error;

    /* Anti-windup */
    if (*I < (-20 << FIXPOINT))
        *I = (-20 << FIXPOINT);
    if (*I > (19 << FIXPOINT))
        *I = (19 << FIXPOINT);

    /* Limit the control signal to valid values */
    if ( u < umin )
        u = umin;
    if ( u > umax )
        u = umax;

    read_lock(&tasklist_lock);
    p = find_task_by_pid(pid);
    if (!p){
        printk(KERN_ALERT "Missing pid\n");
        goto out;
    }

    set_user_nice(p, u >> FIXPOINT);

out:
    read_unlock(&tasklist_lock);
}

```

task to be controlled by the module, the tasks process id (pid) must be known. Also, a reference in the interval [0...100] must be chosen. The command is then supplied to the module by writing it to the mentioned file on the form “ADD pid ref”. If the pid is 8979 and the reference is 5, this can be done with the following line:

```
echo "ADD 8979 5" > /proc/fix/command
```

A task can be unregistered with the command “DEL pid”. The reference can also be changed for an already registered task by adding it again with a new reference (without removing it first). A file `/proc/fix/pid` is created when a new task is registered. This file can be read periodically to receive data about the execution behaviour of the task. The data consists of the registered reference, the period of the sampling, and the execution time received during the last period of the sampling.

2.11 Experiments

All experiments have been performed on the author’s single CPU desktop computer. No special steps were taken before the measurements were made. At the same time as the experiments were made, there were a number of tasks in the system, e.g., X, Firefox, Thunderbird, XEmacs and so on. All bandwidth measurements have been filtered through a moving average window of 4 s. This filtering is done because of the fact that when a task executes, it gets 100% of the CPU-bandwidth and then it gets 0% when it does not execute. Filtering through a moving average window shows the CPU-bandwidth during that window and this is what one wants to achieve in the control.

Running the experiments on a computer with more than one CPU will gain results similar to the ones seen in this section. Except for the fact that there will of course be considerably more load disturbances, as those seen in Experiment 2.

EXPERIMENT 1

The setup consists of four tasks running in endless while-loops as seen in Listing 2.3. Three of the tasks have their nice value set to five while the fourth task’s nice value is used as a control signal to keep its bandwidth at the reference. Somewhere around time 42, the desired reference is changed from 25% to 50%. A plot of the step response can

be seen in Figure 2.7. As can be seen in the plot, the system follows the reference well, but there is considerably more oscillation when the reference is set to 50% than to 25%. This is due to the fact that the reference at 50% makes the control signal iterate back and forth over the non-linear gap in time slice sizes between nice values at 0 and -1 , as can be seen in Figure 2.4. This shows that it is much harder to follow certain references. These hard-to-follow references depend on how many other tasks there are in the system and their respective nice values, so no general rules can be given. \square

EXPERIMENT 2

The setup in this experiment is similar to the one in Experiment 1. However, instead of doing a step in the reference, one of the background tasks is removed at approximately time 100. As can be seen in Figure 2.8, this makes the CPU-bandwidth of the remaining tasks grow and therefore deviate from the desired reference which is set to 25%. The case is handled well by the controller and the disturbance is soon rejected. \square

EXPERIMENT 3

The setup in this experiment is different from Experiment 1 in that two of the tasks are being controlled at the same time. The references for both of the tasks are kept at 25% initially. At time 182, the reference for the first task is changed from 25% to 50%. At around time 320, the reference is changed back to 25%. The result of the step response for the first task can be seen in Figure 2.9. The coupling between the two tasks is visible in Figure 2.10, which shows the disturbance on the second task resulting from the step on the first one. This experiment shows the PWM nature of the control signal and the results of the quantization in the nice value. In Figure 2.9, it can be seen that the control signal is constant both before and after the two steps. But when the reference is set to 50%, the control signal fluctuates a lot. Also note that there is much less oscillation in the CPU-bandwidth when the reference is set to 25% than to 50%. This is due to the fact that some references cannot be kept stationary due to the fact that the nice value is discrete, compare Table 2.1 for more examples. \square

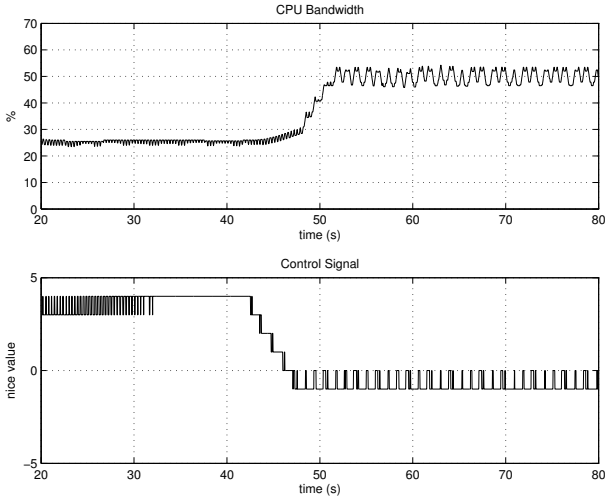


Figure 2.7 Step response of the CPU bandwidth in Experiment 1.

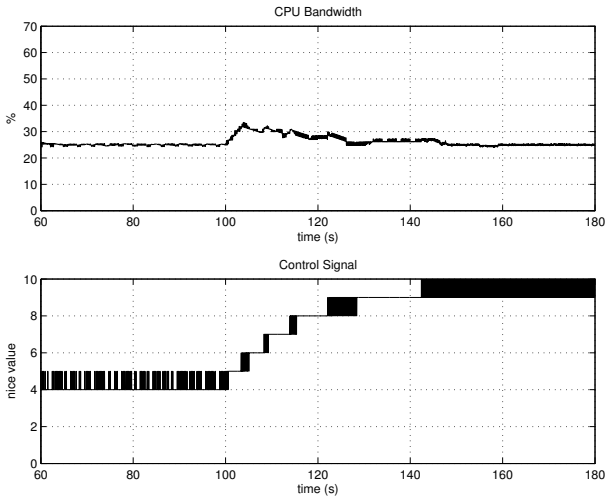


Figure 2.8 Load disturbance of the CPU bandwidth in Experiment 2.

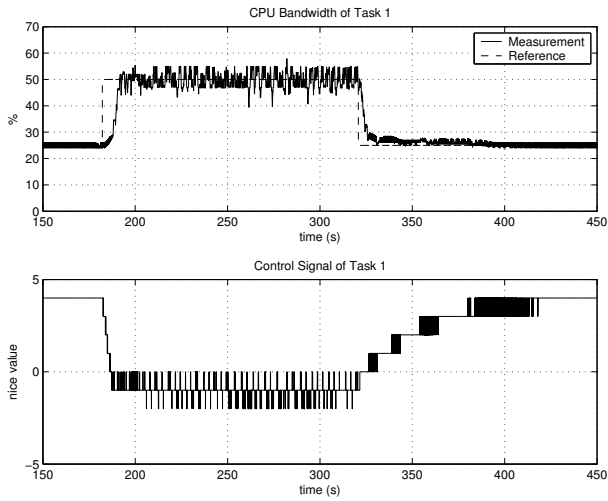


Figure 2.9 Step response of the CPU bandwidth (task 1) when controlling two tasks in Experiment 3.

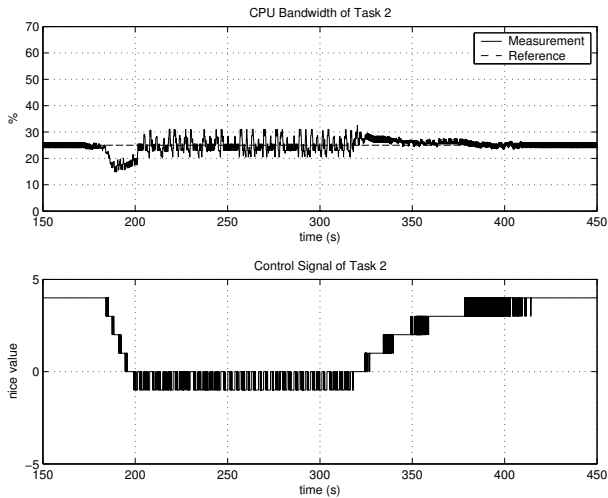


Figure 2.10 Step response of the CPU bandwidth (task 2) when controlling two tasks in Experiment 3.

2.12 Taking the State Into Account

Up until now it has been assumed (for simplicity) that a controlled task is always willing to run, i.e., it is compute-bound. This may be true in some cases, but obviously not in all. Imagine for example that the controlled task is given a reference of 50% but does not need, in fact refuses to use, more than 40% due to the fact that it is waiting on some I/O to occur the rest of the time. The integral part of the PI controller will in this case then add up the difference and increase the control signal in order to remove the error. But as the task itself is unwilling to run and the system can not force it, the difference will remain and the control signal will continue to rise until it hits its limit. This is of course not a good thing, and could be solved by taking the current state of the task into account when controlling it. The strategy could be something like: do not increase the control signal further if the task is not willing to run more. This is more or less an anti-windup scheme, with the difference that the control signal should not be allowed to saturate before it starts to work. This section explains the concept of task states and gives a strategy for solving the above mentioned situation.

Task State

An operating system needs to know what a task is doing in every single moment in order to decide how to schedule it correctly. In Linux, this information is stored in a task's state variable. A task's state is changed many times during its lifetime. The names of the states are more or less arbitrary and vary between different operating systems and literature. Some systems have more fine-grained states, others do not. They are, however, often referred to as:

New The process is being created.

Ready The process is ready to run.

Running The process is running.

Waiting The process is waiting for an event.

Terminated The process is terminated and is waiting to be reaped.

In Linux, the information about the state is saved in the variable `state` in the struct `task_struct` mentioned in Section 2.8. The different states are defined as in Listing 2.5

Listing 2.5 Linux task states taken from `<include/linux/sched.h>`.

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_STOPPED          4
#define TASK_TRACED           8
#define EXIT_ZOMBIE           16
#define EXIT_DEAD              32
```

If the state is set to `TASK_RUNNING`, it means that the process is ready to be run. But it does not necessarily mean that the process is running at this very moment. This “Running” state is the same as the normal “Ready” state mentioned earlier. `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE` are “Waiting” states. Interruptible means that the process can be sent a signal that wakes it up before its requested sleep time is over. `TASK_STOPPED` means that the process is stopped, often because it was sent a signal. This is also a kind of wait state. `TASK_TRACED` is used for debugging. `TASK_ZOMBIE` is set when a process has terminated but has not yet had its status collected by the parent. `EXIT_DEAD` is also used in the termination phase.

Strategy

The idea to update the control signal only if the task is willing to run, sounds good at first. It is, however, not as simple as it first might seem. The obvious question to answer is: how do we know if a task is willing to run more than it already does? The idea used in the current implementation is to sample the state of the task at the same time as the execution time. The controller is then only executed if two consecutive samples show that the task is in the `TASK_RUNNING` state. This strategy works well if the task is usually in the `TASK_RUNNING` state for a longer time than the time between two consecutive samples of the controller. How long time a task spends in its running state depends highly on its own workload during a certain time interval, but also on

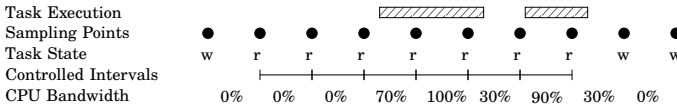


Figure 2.11 Example of how the strategy for controlling non-compute bound tasks works.

the other tasks in the system as the task might get interrupted by a higher priority task. This makes it hard to give any general rules and hence draw any conclusions to be used for more accurate control.

Why does the Strategy work?

The reason why the presented strategy works, i.e., the control signal does not saturate, is the following: A task with a low priority will be in the TASK_RUNNING state for a long time. This is due to the fact that it will be preceded and interrupted by tasks with higher priorities. It will not switch from the TASK_RUNNING state until it has finished its current work load. If the priority of the task is increased, the task may not be preceded by as many tasks as before and it will also not be interrupted by as many. Hence, it will finish earlier and therefore be a shorter time in the TASK_RUNNING state. In essence, a high priority gives a short ready time. As the execution time demand is constant, the ratio between executed time and time spent in the TASK_RUNNING state will increase if the priority is increased. At a certain point, an equilibrium will be reached, where the reference is met during the period when the task is in the TASK_RUNNING state, and hence the control signal will be constant.

Example Figure 2.11 shows the sampling points of the controller and the task’s state at those points. It also shows the task’s execution trace and which of the sample intervals that are used by the controller. An *r* in the figure means that the task is in the TASK_RUNNING state, and a *w* that it is in one of the “Waiting” states. During the controlled intervals, the ratio between used time and time spent in the running state is approximately 48%.

EXPERIMENT 4

This experiment consists of one periodic task that executes for approximately 40 ms and then sleeps for 60 ms repeatedly. This results in a task that uses at most 40% of the CPU even if it is alone in the system. Controlling such a task with the method explained in Section 2.8 would make the control signal saturate, because the error will never go away if a reference larger than 40% is given. Two load tasks of the same type as used in Section 2.11 with `nice` values at 5 are also present in the system.

As can be seen in Figure 2.12, the proposed scheme works well in practise. In the beginning of the plot, the reference is higher than can be achieved, and at time 55 it is set to an even higher value, but the control signal still behaves well. It can also be seen that the controller is still able to follow reference changes when they are lower than 40%.

The observant reader may notice the delay and the following under-shoot at time 160. Also note that this behaviour does not show up at any of the other step changes in the plot. This is not an integrator windup as might first be thought, but is instead due to the fact that the system has marked the task as interactive and therefore given it an additional bonus. When the task after some time is marked as non-interactive, it loses its bonus and this results in the under-shoot. This is the same thing as stitching in a mechanical system.

The fact that non-compute-bound tasks can be controlled using the presented strategy, significantly increases the usability of the technique. It opens up a new range of applications to be controlled, such as event-driven tasks that need some percentage of the CPU when they want to execute, but otherwise sleeps most of the time. This is the case with mp3-decoders and video-players, but also with web-servers and many other applications. \square

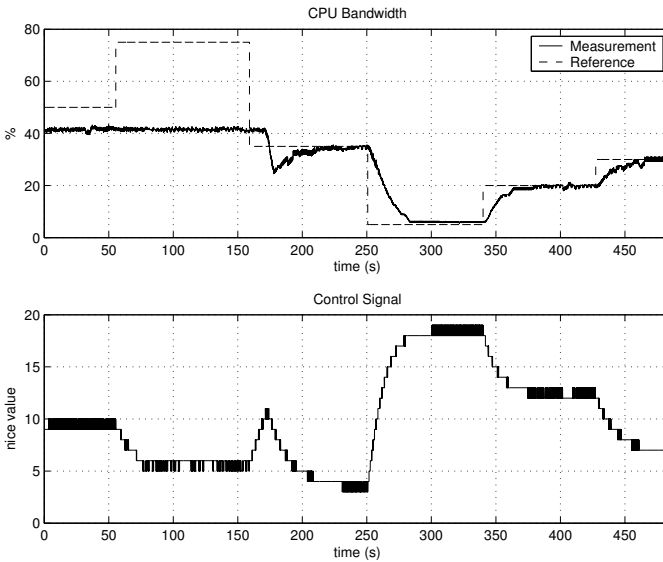


Figure 2.12 Step responses for a non compute-bound task when taking the task’s state into account in Experiment 4.

2.13 Related Work

Reservation based scheduling is not a new concept and has been around in one form or another for many years. The concept has been called *fair-share scheduling* [Essick, 1990; Kay and Lauder, 1988; Henry, 1984] but is also known under the name *proportional-share scheduling* [Fong and Squillante, 1995; Stoica and Abdel-Wahab, 1995; Waldspurger and Weihl, 1995b; Waldspurger and Weihl, 1995a]. A good summary of this field, together with more details can be found in [de Jongh, 2002]. In the real-time systems area, a similar concept is known under the name Constant Bandwidth Server (CBS) [Abeni and Buttazzo, 1998]. An implementation of a CBS running in the Linux 2.4.18 kernel can be found in [Abeni and Lipari, 2002]. The mentioned implementation works by raising the priority of the task that it currently wants to run to the highest possible value. A comparison between the constant

bandwidth allocation and the proportional share allocation methods can be found in [Abeni *et al.*, 1999]. Other interesting related topics in the real-time area is the Resource-Kernel [Rajkumar *et al.*, 1998] and the Sporadic Server [Sprunt *et al.*, 1989].

The idea of using the nice value as a way to enforce CPU fractions has been known before. One early implementation is the *Watson Share scheduler* [Moruzzi and Rose, 1991], implemented on top of a standard AIX operating system at the Compute Power Server Cluster at IBM. It is also mentioned in [Hellerstein, 2004] and [Hellerstein *et al.*, 2005] as something that in UNIX can be done in theory, but is complicated in practice because of the non-linear relationship between nice, the number of processes and the CPU fraction received. Provided that the number of jobs in the system is fixed, and that they are all present from the same time and onward, a deterministic analysis of the steady state shares is possible. [Hellerstein, 1993] shows how this can be used to statically calculate the base priorities on a uniprocessor in the presence of *decay-usage scheduling* in UNIX. [Epema, 1998] extends this analysis to the multiprocessor case.

An interesting Linux kernel project in this area is *Class-based Kernel Resource Management (CKRM)* [CKRM, 2006] and [Nagar *et al.*, 2004] which aims at providing differentiated service to resources such as CPU time, memory pages, I/O and incoming network bandwidth. It accomplishes CPU reservations by scaling the `time_slice` value and re-queuing tasks. Parts of this project is used in “SuSE Linux Enterprise Server 9”, but not the CPU controller.

2.14 Conclusions

An exposition of the Linux 2.6 scheduler has been done and a method for controlling the CPU bandwidth given to tasks in Linux has been presented. The presented method has been shown to work, both for compute-bound and non compute-bound tasks. A number of experiments have been performed in order to show that the technique works in reality.

3

A Simulation Tool for Wireless Control

3.1 Introduction

Sensor/actuator networks and mobile robots are application areas for embedded real-time systems where wireless communication plays a vital role. The computing and communication resources in these types of applications are often severely limited, making integrated design approaches important. Another common characteristic of these systems is that they interact with their environment. One example is a sensor network that monitors the presence of moving objects in some environment. Other examples are mobile robots moving around in the environment or sensor/actuator networks that implement networked control loops.

Simulation is a powerful technique that can be used at several stages of system development. In order to support the applications at hand, co-simulation facilities are crucial. It should be possible to simultaneously simulate the computations that take place within the nodes, the wireless communication between the nodes, the sensor and actuator dynamics, the dynamics of the mobile robots, and the dynamics of the environment, including the physical systems under control. In order to model the limited resources correctly, the simulation model must be quite realistic. For example, it should be possible to simulate

the timing effects of context switches and interrupts in a multi-tasking real-time kernel implementing the control logic of the nodes. It should also be possible to simulate the effects of collisions and contention in the wireless MAC layers. Due to simulation time and size constraints, it is at the same time important that the simulation model is not too detailed. For example, simulating the computations on a source code level, instruction for instruction, would be overly costly in most cases. The same applies to simulation of the wireless communication at the radio interface level or on the bit transmission level. In some cases however simulation on this low level is exactly what one wants to achieve, and then products like, e.g., Simics [Magnusson *et al.*, 2002] are useful.

There are a number of simulation environments available for networked systems, see the related work section at the end of this chapter for an overview. However, the majority of these only simulate the wireless communication and the node computations. Hence, something more is needed. TrueTime [Cervin *et al.*, 2003; Andersson *et al.*, 2005] is a co-simulation tool that is being developed at Lund University since 1999. By using TrueTime it is possible to simulate the temporal behavior of computer nodes and communication networks that interact with the physical environment. The network support in the early releases of the tool was restricted to wired networks. In this chapter the wireless network block, available in recent versions, is presented together with the means to simulate battery-powered nodes, local clocks and Dynamic Voltage Scaling (DVS). This, in combination with the already present features, make it possible to concurrently simulate all the aspects described above. This opens up a wide range of new application types for simulation, e.g., teams of collaborating or competing mobile robots interacting with their environment. Another example could be sensor networks with mobile or stationary nodes communicating via wireless ad-hoc networks internally and through a gateway node to back-end servers using wired networks.

3.2 Wireless Networks

A wireless network is different from a wired one in many ways. Most of the differences between them rise from the fact that in a wired network, the signal follows a guarded medium from the sender to the

receiver. The signal has therefore more or less the same characteristics at the two points. In a wireless network, the case is the opposite. The signal may have travelled through a number of different matters on its way from the sender to the receiver. It may also have taken a number of different paths. This results in a signal which differs significantly between the two points. This section will give a very condensed description of some important properties of wireless networks.

Wireless networks can be divided into two groups, infrastructure based networks and ad-hoc networks. Infrastructure based networks are very common in WLANs and mobile phone systems. Typically all communication takes place between nodes and access points and there is no direct communication between nodes. This makes it easy for the access point to control the medium access. It is also rather straightforward to let the access point perform routing between different networks when the node is not within the access point's own signal reach. The drawback is that infrastructure based networks are not so flexible. Ad-hoc networks on the other hand are very flexible, and do not need any infrastructure at all to work. In such networks, nodes can communicate directly with each other and hence there is no need for access points. This on the other hand makes routing a bit harder because no central intelligence exists.

In a wireless network, radio waves are often used as the physical transportation layer for the signal, and then an antenna is needed. IEEE 802.11 also supports Infrared (IR) light as a medium, but it is hardly used in any commercial products. There exists a number of different antenna models for radio. One often used theoretical model of an antenna is the isotropic one. This is simply a single point in space which radiates equal amounts of power in all directions. In reality though, all antennas have a more or less directive effect and many antennas are even designed in order to get such effects.

It is common that radio stations are built such that they can not send and receive signals at the same time. As a consequence of this, it is not possible for a sending station to hear if another station starts to send during its own transmission. Duplex stations can of course be built, but the technology is a little more complex and expensive.

When simulating a wireless network, it is important to take the path-loss into account. The signal power is always much lower in the receiver node than in the sender node. Parts of this comes from the

fact that wireless devices do not have a guarded medium, and therefore the radio signal power is spread in all or some directions depending on the antenna being used. Many obstacles such as buildings, trees, fog, snow and rain will also decrease the signal power level in the receiver. Due to large obstacles, there can be an extreme form of attenuation called blocking or shadowing, which makes it impossible to receive the radio signal at all. The signals may also reflect on objects such as buildings or mountains. This reflection makes it possible for the radio signal to take many different paths from the sender to the receiver, i.e., multi-path propagation. If the sender and the receiver are moving, then the channel characteristics will change over time. Because of the multi-path propagation, the received signals may then sometimes have different phase and cancel each other. The power level can therefore change significantly even if the nodes are moved only a small distance. This is often referred to as short-term fading. Long-term fading is the fact that the signal fades with distance.

In a wireless network, the air can be considered as a shared medium and therefore interference from other terminals must be taken into account in the simulation. It may also happen that nodes that are not able to hear each other when they are sending, still disturb each others transmissions in a receiving node situated in between them. This situation is often referred to as the hidden node problem.

3.3 The TrueTime Simulator

TrueTime is based on Simulink [The Mathworks, 2001], the graphical simulation environment of MATLAB, and consists of computer, battery and network blocks as shown in the block library in Figure 3.1. The TrueTime blocks are connected with ordinary Simulink blocks to form a real-time control system. The main feature of TrueTime is the possibility of co-simulation of the interaction between the real-world continuous-time dynamics and the computer architecture in the form of task execution and network communication.

The TrueTime computer block executes user-defined tasks and interrupt handlers representing, e.g., I/O tasks, control algorithms, and network drivers. The scheduling policy of individual computer blocks is arbitrary and decided by the user. Execution times of tasks and inter-

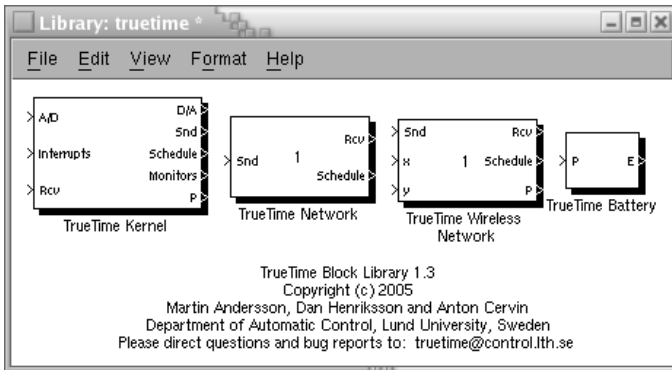


Figure 3.1 The TrueTime block library.

rupt handlers can be modeled as constant, random, or data-dependent. Furthermore, TrueTime allows simulation of context switching and task synchronization using events or monitors.

The TrueTime network blocks distribute messages between computer nodes according to a chosen network model. Communication models supported by the wired network block are: CSMA/CD (e.g. Ethernet), CSMA/AMP (e.g. CAN), Round Robin (e.g. Token Bus), FDMA, TDMA (e.g. TTP), and Switched Ethernet. Only packet-level simulation is supported – it is assumed that higher protocol levels in the kernel nodes have divided long messages into packets, etc. Only the properties related to the communication timing are modeled, i.e., the simulation is not performed on bit level.

To illustrate the TrueTime simulation capabilities, a small example is given in Listings 3.1, 3.2 and 3.3. The example creates an interrupt handler which is triggered when a message arrives on the network. The message is read and then directly put in a mailbox. A task instance is created in order to handle the message outside the interrupt handler.

The focus of this chapter is on the wireless network modeling and simulation. For more details on the computer and wired network blocks, see [Cervin *et al.*, 2003; Andersson *et al.*, 2005]. TrueTime 1.3 is available for download at <http://www.control.lth.se/user/dan.henriksson/>.

Listing 3.1 Initialization of the TrueTime kernel.

```

function example_init(argument)
% Initialize the TrueTime kernel
ttInitKernel(0, 0, 'prioFP'); % nbrOfInputs, nbrOfOutputs, fixed priority

% Create a mailbox
ttCreateMailbox('network_messages', 10) % mailboxname, maxsize

data.msg=[];
% Create a task
% name, deadline, priority, codefunction, data structure
ttCreateTask('network_response_task', 100, 3, 'taskcode', data);

% Create a network interrupt handler
% name, priority, codefunction
ttCreateInterruptHandler('nw_handler', 3, 'handlercode');
% Initialize the network
ttInitNetwork(1, 'nw_handler');          % nodenumber, handlername

```

Listing 3.2 Task code.

```

function [exectime, data] = taskcode(seg, data)
switch seg,
case 1,
    data.msg = ttTryFetch('network_messages'); % Read a message from a mailbox
    exectime = 0.00002;                       % execution time
case 2,
    % Reply to the sender
    ttSendMsg(2, data.msg, 10);               % receiver, data, length
    exectime = -1;                            % finished
end

```

Listing 3.3 Code for the network interrupt handler.

```

function [exectime, data] = handlercode(seg, data)
temp = ttGetMsg;                             % Read a message from the network
ttTryPost('network_messages', temp); % Put the message in a mailbox
ttCreateJob('network_response_task'); % Create an instance of a task
exectime = -1;

```

3.4 The TrueTime Wireless Network

This section will describe the wireless network modeling, the implementation of the IEEE 802.11b/g WLAN and IEEE 802.15.4 ZigBee standards, and the interface of the wireless block within TrueTime. Again it should be noted that the scope of the wireless block (and the TrueTime environment as a whole), is to simulate the main properties related to the timing characteristics and not to be a complete wireless network simulator. Consequently, the simulation of network messages is not done on the bit or radio interface level.

Modeling of the IEEE 802.11b/g Protocol

The TrueTime wireless network block simulates medium access and packet transmission, i.e., the medium access control (MAC) sub-layer of the IEEE 802.11 reference model. 802.11b/g is used in many laptops and mobile devices of today, and is because of its heavy use a good candidate to include in a simulator. The 802.11b standard is quite complex and is described in [IEEE, 1999a] and [IEEE, 1999b]. A more condensed presentation can be found in [Schiller, 2003]. The MAC schemes of 802.11b/g are the same as in ordinary 802.11, therefore the simulation model presented here is valid for 802.11 as well. The presented model captures the following aspects:

- Direct sequence spread spectrum (DSSS¹) physical layer (PHY). Other non-supported PHY layers of 802.11 are frequency hopping spread spectrum (FHSS) and infra-red (IR).
- Ad-hoc wireless networks, as opposed to infrastructure-based ones.
- Isotropic antenna.
- Inability to send and receive messages at the same time.
- Path loss of radio signals modeled as $\frac{1}{d^a}$ where d is the distance in meters, and a is a parameter chosen to model the environment.
- The mandatory basic access method based on CSMA/CA.
- Interference from other terminals (shared medium).

¹Each information bit in the message is XOR'ed with a sequence of 0's and 1's in order to make the signal more insensitive to, e.g., multi-path propagation

The implemented simulation model does not capture very detailed or compute intensive aspects such as shadowing, reflection or multi-path propagation. Many of these features can however be implemented in the user-defined path-loss function described in section 3.6.

A package transmission is modeled like this: The node that wants to transmit a packet checks to see if the medium is idle. The transmission may proceed, if the medium is found to be idle, and has stayed so for a time specified in the standard ($50 \mu\text{s}$ when using DSSS). A random back-off time is chosen and decremented in the same way as when colliding (described in the end of this section), if the medium is found to be busy. When a node starts to transmit, its relative position to all other nodes in the same network is calculated, and the signal levels in all these nodes are calculated according to the path-loss formula $\frac{1}{d^a}$.

The signal is assumed to be possible to detect, if the signal level in the receiving node is larger than a configurable threshold (receiver signal threshold). If this is the case, then the signal-to-noise ratio (SNR) is calculated and used to find the block error rate (BLER). Note that all other transmissions add to the background noise when calculating the SNR. The BLER, together with the size of the message, is used to calculate the number of bit errors in the message and if this number is lower than another threshold (error coding threshold), then it is assumed that the channel coding scheme is able to fully reconstruct the message. If there are (already) ongoing transmissions from other nodes to the receiving node and their respective SNRs are lower than the new one, then all those messages are marked as collided. Also, if there are other ongoing transmissions, which the currently sending node reaches with its transmission, then those messages may be marked as collided as well depending on the signal strength in the receiving nodes.

Note that a sending node does not know if its message is colliding, therefore ACK messages are sent on the MAC protocol layer. From the perspective of the sending node, lost messages and message collisions are the same, i.e., no ACK is received. If no ACK is received during a certain configurable time, the message is retransmitted after waiting a random back-off time within a contention window. The contention window size is doubled for every retransmission of a certain message. The back-off timer is stopped if the medium is busy, or if it has not been idle for at least a time specified by the protocol ($50 \mu\text{s}$ when using DSSS). There are only a certain number of retransmissions before the sender gives up on the message and it is not retransmitted anymore.

Modeling of the IEEE 802.15.4 Protocol

ZigBee is a protocol designed with sensor and simple control networks in mind. It has a rather low bandwidth, 250 Kb/s, but also a really low power consumption. Although it is based on CSMA/CA as 802.11b/g, it is much simpler and the protocols are not very similar.

The packet transmission model used for simulating ZigBee is similar to WLAN, but the MAC procedure differs and is modeled as:

1. Initialize:
NB=0
BE=macMinBE
2. Delay for a random number of backoff periods in the interval $[0, 2^{BE} - 1]$
3. Is the medium idle?
if yes: send
else: goto 4
4. Update the backoff counters:
NB=NB+1
BE=min(BE+1, aMaxBE)
5. Is NB>macMaxCSMABackoffs?
if yes: drop the packet
else: goto 2

The variable names are taken from the standard to make comparisons easier. A small explanation of their names is provided below.

NB Number of backoffs.

BE Backoff exponent.

macMinBE The minimum value of the backoff exponent in the CSMA/CA algorithm. The default value is 3.

aMaxBE The maximum value of the backoff exponent in the CSMA/CA algorithm. The default value is 5.

macMaxCSMABackoffs The maximum number of backoffs the CSMA/CA algorithm will attempt before declaring a channel access failure. The default value is 4.

Calculation of Error Probabilities

During the calculation of error probabilities, it is for simplicity assumed that BPSK² is always used in the transmissions. This is of course an approximation, but it relates well to reality.

Assume that a symbol is sent, in our case this is a bit, i.e., a 0 or a 1. Additive white Gaussian noise gives a probability density function, for the received symbol, that for some signal-to-noise ratio will look like Figure 3.2. A threshold is then used to decide if the received symbol is a 0 or a 1. The decision threshold is marked as a line in the middle of the figure. The darker area to the left of the threshold gives the probability of a symbol error. A higher signal to noise ratio translates the curve to the right, making the probability of error smaller.

The above standard procedure should ideally be performed for every bit in the message. The total number of calculated bit errors should then be compared to the error coding threshold. This is, however, not done, because it would computationally be very expensive. Instead, the maximum noise level during the transmission is saved, and used to calculate the worst case SNR. By assuming that bit errors in a message are uncorrelated, it is deduced that the number of bit errors, X , belongs to a binomial distribution $X \in Bin(n, p)$. Where n is the number of bits in the message, and p is the probability that a certain bit is erroneous. If the value of n is large, the binomial distribution can be approximated with a normal distribution, using the central limit theorem. This gives that $X \in N(np, \sqrt{npq})$ where $q = 1 - p$. What we are really interested in is the probability that bn , where b is the error coding threshold, is larger than the total number of bit errors in a message. This probability is calculated using

$$P(X \leq bn) = \begin{cases} \Phi\left(\frac{bn - np}{\sqrt{npq}}\right) & \text{if } bn - np > 0 \\ 1 - \Phi\left(\frac{|bn - np|}{\sqrt{npq}}\right) & \text{if } bn - np \leq 0 \end{cases}$$

where Φ is the standard normal cumulative distribution function.

²Binary Phase Shift Keying (BPSK) is a means of transmitting symbols by altering the phase of a reference signal. It uses two phases separated by 180° and is hence binary.

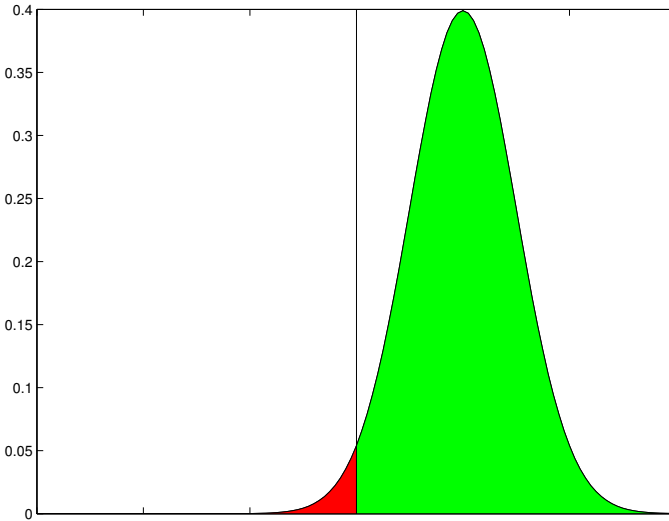


Figure 3.2 Probability density function for a received symbol when using binary phase shift keying and additive white Gaussian noise. The line in the middle is the decision threshold. The area to the left of the threshold gives the probability of an erroneous decision. The area to the right gives the probability of a correct decision.

Example Assume that a message consists of 100 bits, i.e., $n = 100$. The probability that a certain bit is erroneous has been calculated to 0.1 using the above method, i.e., $p = 0.1$ and $q = 1 - p = 0.9$. The error coding threshold has been set to 5%, i.e., $b = 0.05$. Then the probability that we can decode the complete message is

$$P(X \leq bn) = 1 - \Phi\left(\frac{|bn - np|}{\sqrt{npq}}\right) = 1 - \Phi\left(\frac{5}{\sqrt{9}}\right) \approx 0.0478$$

The TrueTime Wireless Block Interface

The structure of the wireless block as seen from a user point of view is very similar to the wired network block. The main difference between the wired network block and the wireless is that the x and y coordinates of the simulated computers are made available to the wireless block with two vector input ports as seen in the block library in Fig-

ure 3.1. These input coordinates give the position of the nodes and may therefore change over time if the nodes are moving. A node in this context is a unit consisting of a computer block connected to a wireless block, and optionally some dynamics. In the current implementation, nodes can only move in a two-dimensional space. It is however straightforward to add a third dimension, if needed in future simulations.

The wireless block is event-driven, i.e., when a node tries to transmit a packet, the packet is put in an input buffer and a trigger signal is sent to the block on the Snd port. When the simulated transmission of the package is finished, the packet is put in a buffer at the receiving node and a new trigger signal is sent to the receiving node on the Rcv port. A packet contains information about the sending and receiving computer node, arbitrary user data, and the packet length.

The wireless network block is configured through the block mask dialog, see Figure 3.3. The available options consist of:

Network type. Determines the MAC protocol to be used. Can be either 802.11b/g (WLAN) or 802.15.4 (ZigBee).

Network number. A unique identifier for the wireless network block. Useful when nodes are connected to multiple networks.

Number of nodes. Number of nodes connected to the network. Affects the size of Snd, Rcv and Schedule inputs and outputs of the wireless block.

Data rate. Data transmission speed of the network in bits/s.

Minimum frame size. Messages shorter than this will be padded.

Transmit power. The signal strength used when transmitting a message. According to the standard, this is limited to a maximum of 1000 mW in USA, 100 mW in Europe and 10 mW in Japan.

Receiver signal threshold. If the received signal strength is above this value, then the medium is regarded as busy.

Path-loss exponent. The number a in the path-loss formula $\frac{1}{d^a}$.

Special pathloss function. Gives the user the possibility to implement his/her own path-loss function in a MATLAB m-file.

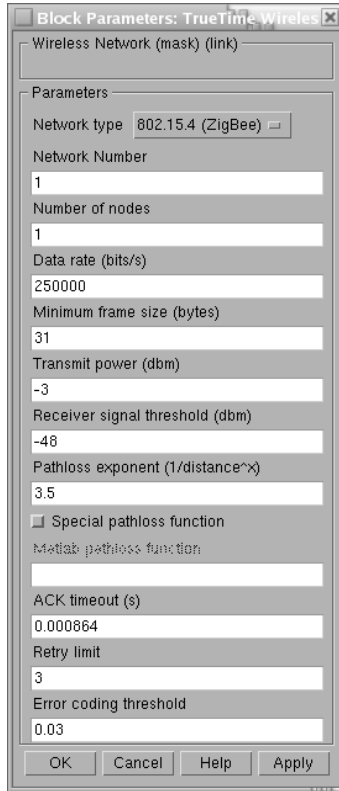


Figure 3.3 The dialog of the TRUETIME wireless network block.

ACK timeout. The time a sending node will wait for an acknowledgement before retransmitting a message.

Retry limit. The maximum number of times a node will try to retransmit a message before giving up.

Error coding threshold. A number in the interval $[0, 1]$ which defines the percentage of block errors in a message that the coding can handle. For example, certain coding schemes can fully reconstruct a message if it has less than 10% block errors.

Some of the network parameters can also be set on a per node basis with the command `ttSetNetworkParameter`. This function makes it possible to dynamically change these network parameters. At the moment the following parameters are supported:

transmitpower Signal strength used when transmitting a message.

predelay The time a message is delayed by the network interface at the sending end. This can be used to model, e.g., a slow serial connection between the computer and the network interface.

postdelay The time a message is delayed by the network interface at the receiving end.

The default parameter value is 0 for the `predelay` and the `postdelay`. The `transmitpower` parameter is only valid when using the wireless network and defaults to whatever is set in the block mask dialog.

3.5 Implementation Details

This section will give a brief description of how the event-based implementation of the wireless network block exploits the zero-crossing detection mechanism in Simulink.

The implementation uses two callback functions to interact with Simulink: The `mdlOutputs` (Listing 3.4) and the `mdlZeroCrossings` (Listing 3.5). `mdlOutputs` is called by Simulink at each simulation time step, to compute the block's output signal and store it in the S-function's output signal arrays. `mdlZeroCrossings` is used by Simulink to find discontinuities, in order to insert additional simulation time steps before and after the time they occur. The `mdlZeroCrossings` function should be implemented in such a way that it returns a value that crosses zero at the time of a discontinuity.

When a message is sent on the network, the `Snd` input port of the network block is changed as a step. This creates a discontinuity which leads to the fact that `mdlOutput` is executed a small time before, exactly at the time of the discontinuity, and also a small time after it. This makes sure that the block is always executed when a message is sent. The other type of event that the block should take care of is internal events. These are generated when the transmission of a message is

finished and the result should be propagated to a receiving node. Internal events are also generated, e.g, when a backoff counter reaches zero. These internal events are taken care of in the `mdlZeroCrossings` function. Every time the network function is run, it saves the time of the next internal event to occur in a variable `nwsys->nextHit`. When the `mdlZeroCrossings` function is executed, it returns the difference between the time of the next event and the current time. It also returns zero if it detects that the input signals have changed value since the last invocation.

A small example of the timing is given in Listing 3.6. The example consists of a Simulink step block connected to an S-function. When the mentioned callback functions are executed, the S-function prints out the current time. The step has been set to occur at time 1.5. As can be seen in Listing 3.6, `mdlZeroCrossings` detects the change in the input signal at time 1.5. `mdlOutput` is, however, not executed with the new input signal until time $1.5 + \epsilon$. This means that there is usually a small offset from when the input changes to when the block is executed with the new input signal. The size of the time delay depends on which solver that has been chosen in the simulation. The time delay is, however, very small and in the presented example it was $2.8 \cdot 10^{-14}$ s.

The `runNetwork` function, called from the code in Listing 3.4, is responsible for transmitting messages according to the chosen protocol. This is done in the following way: When a new message arrives, it is put in a queue called `preprocQ`. Messages are also put in this queue if they have collided and must wait a certain time before they are transmitted again. The message waits in this queue until the `preDelay` or the backoff time has expired. The message is then transmitted if the chosen protocol admits it. When a message is in transmission, it is moved to another queue called `inputQ`. If the transmission was successful, the message is moved to the `outputQ`, otherwise it is put in the `preprocQ` again with a suitable backoff time. When the message has waited for the specified `postDelay`, it is moved to the `postprocQ` where the receiving node can access it. One problem with the current implementation is that it only remembers the time of the next event. A better solution would be to calculate the time of all known future events and save them in a data structure. This data structure could then be updated when new events arrive.

Listing 3.4 Pseudo-code for the mdlOutputs function.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    if ( any input has changed value ) {
        /** Run the network code, and compute when the next invocation
         * should occur.
         */
        nwsys->nextHit = runNetwork();
    }
    Update all output signals;
}
```

Listing 3.5 Pseudo-code for the mdlZeroCrossings function.

```
static void mdlZeroCrossings(SimStruct *S)
{
    Read and store all inputs;

    if ( any input has changed value ) {
        nwsys->nextHit = ssGetT(S);
    }
    /**
     * Tell Simulink when we would like to run next. This may be now,
     * or sometime in the past or future.
     */
    ssGetNonsampledZCs(S)[0] = nwsys->nextHit - ssGetT(S);
}
```

Listing 3.6 A small example showing how a discontinuity in the input signal at time 1.5 creates additional simulation time steps. The additional steps are $1.5 - \epsilon$, 1.5 and $1.5 + \epsilon$.

```
mdlOutputs:          0.000000000000000000e+00
mdlZeroCrossings:    0.000000000000000000e+00
mdlZeroCrossings:    1.000000000000000000e+00
mdlOutputs:          1.000000000000000000e+00
mdlZeroCrossings:    1.000000000000000000e+00
mdlZeroCrossings:    2.000000000000000000e+00
mdlZeroCrossings:    1.500000000000000000e+00
mdlZeroCrossings:    1.250000000000000000e+00
mdlZeroCrossings:    1.4999999999997157829e+00
mdlOutputs:          1.4999999999997157829e+00
mdlOutputs:          1.500000000000000000e+00
mdlZeroCrossings:    1.500000000000000000e+00
mdlZeroCrossings: input has changed value at 1.500000000000000000e+00
mdlZeroCrossings:    2.000000000000000000e+00
mdlZeroCrossings:    1.750000000000000000e+00
mdlZeroCrossings:    1.50000000000002842171e+00
mdlOutputs:          1.50000000000002842171e+00
mdlZeroCrossings:    1.50000000000002842171e+00
mdlZeroCrossings:    2.000000000000000000e+00
mdlOutputs:          2.000000000000000000e+00
mdlZeroCrossings:    2.000000000000000000e+00
mdlZeroCrossings:    3.000000000000000000e+00
mdlOutputs:          3.000000000000000000e+00
mdlZeroCrossings:    3.000000000000000000e+00
```

3.6 Other New Simulation Features

Local Clocks

Having local independent clock representations with respect to a global time, will facilitate simulation of many problems related to clock drift in asynchronous systems and failure in the clock synchronization for synchronized systems.

The local clocks are manipulated by specifying time offsets and drifts, which is typically done during the initialization phase for each node. An ideal clock is obtained by setting these values to zero. The timing primitives in TrueTime that are affected by the change in clock representation are; `ttSleep`, `ttSleepUntil`, `ttCurrentTime`, and `ttCreateTimer`, which all operate on the local clocks in the node.

Power Consumption and Dynamic Voltage Scaling

The power constraints of networked embedded systems make it desirable to model power consumption. Therefore, models of power devices (batteries) have been incorporated into the TrueTime framework. This makes it possible to model and evaluate power management strategies at different levels of the system for energy-efficient implementations. This includes dynamic voltage scaling algorithms, energy-aware scheduling and communication protocols. As a means for this, TrueTime supports dynamic changes of the CPU speed in the different nodes, and the possibility to set a nominal power consumption.

Each node has its own battery block, which is connected in a feedback loop with the computer block. The inputs to the battery block could be any form of power consumption, e.g., CPU, network transmission and receiving activity, and sensors and actuators. The battery itself is modeled as a simple integrator, fed by the sum of the battery inputs. The output of the battery is fed to the computer block, and as this level drops to zero all task execution is stopped. The execution will continue if the battery is recharged.

User-Defined Path-Loss Function

The default path-loss function (or propagation model) used in the TrueTime wireless simulations is

$$P_{receiver} = \frac{1}{d^a} P_{sender}$$

where P is the power, d is the distance in meters, and a is a parameter that can be chosen to model different environments. This model is often used in simulations, but in some cases it can be advantageous to use other models. Therefore, TrueTime has the possibility to register a user-defined path-loss function. The function is written as a MATLAB m-file and can therefore take advantage of all the built-in functions available in MATLAB or Simulink. This includes in particular the possibility to use persistent variables, i.e., variables which are retained in memory between calls to the function. This function can, for example, be used to model a Rayleigh³ fading or blocking of radio signals to and from certain points in the environment. At the moment, nodes in the TrueTime framework only have x and y coordinates, but if a direction was to be introduced this function could also be used to model directive effects in the antenna behaviour.

The MATLAB function takes the following arguments

- Transmission power
- Name of the sending node
- x and y coordinates of the sending node
- Name of the receiving node
- x and y coordinates of the receiving node
- Current simulation time

and returns the signal power in the receiving node.

A small example showing the structure of how a Rayleigh fading could be implemented can be seen in Listing 3.7.

³In a Rayleigh fading, the relative speed of two nodes and the number of multiple paths that the signal takes from the sender to the receiver is taken into account. See Figure 3.4 for an example.

Listing 3.7

```

function [power]=rayleigh(transmitPower, node1, x1, y1, node2, x2, y2, time)

% Calculate the exponential pathloss
distance = sqrt((x1 - x2)^2 + (y1 - y2)^2);
power = transmitPower/(distance+1)^3.5;

% Kalman filter to get the relative velocity of the two nodes
velocity = kalman_velocity(node1, x1, y1, node2, x2, y2, time);
% Calculate the rayleigh fading
factor = calculate_rayleigh(node1, node2, velocity, time);

% Add the rayleigh fading to the exponential path loss
power = power * factor;

```

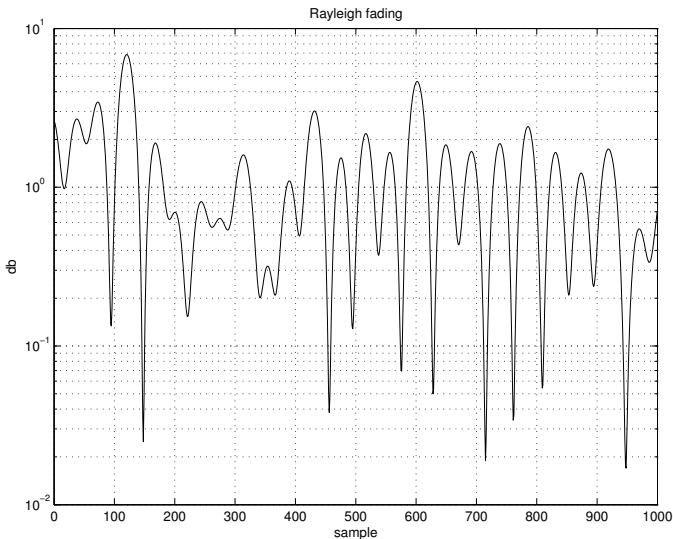


Figure 3.4 Example of a Rayleigh fading using a radio frequency of 2.4 MHz. Nodes moving with a relative speed of 6 km/h, 1 second sampling interval and 10 random phasers.

3.7 Simulation Case Studies

The TrueTime simulation environment for networked embedded systems will now be demonstrated in two simulation scenarios. The first simple scenario is intended to demonstrate the wireless communication model and the impact of interfering nodes and physical locations on the communication timing. The other example is more elaborate and show the use of TrueTime in a research area that is currently receiving much attention in the real-time systems community, distributed control of mobile agents.

A Simple Communication Scenario

This setup contains four nodes communicating over a wireless network. The example is intended to show how the distance and therefore the path-loss between sender and receiver, and also interference from other sending nodes affect the communication timing.

Two situations will be described with the geographic locations of the nodes displayed in Figure 3.5. In the first setup (labeled a) in the figure), node 1 is far away from the others and therefore its signal level will be too low to receive in the other nodes. This also means that node 1 will not be able to disturb the transmissions of the other nodes. The circle around each node denotes the distance at which the power of the original signal has been reduced to the threshold value. In this simulation, the transmit power was 100 mW, the threshold 2 mW and the path loss exponent 2. In the second setup, node 1 has moved considerably closer and is now within the reach of all other nodes.

In both setups, nodes 1, 2, and 3 send periodic messages to the receiver node (node 4). Closeups of the resulting network schedules can be seen in Figure 3.6. The network schedule has the following representation. Low means that the node is idle and has nothing to send. Medium corresponds to that the node is waiting to transmit a message, but for some reason has not started yet, i.e., the net is busy or it is counting down its back-off timer from a previous collision. And finally, high means that the node is sending a message. Note that a node does not know that its message has collided until it does not receive an ACK message from the receiver. Therefore, a transmission does not end until the complete message has been sent, even if it collides.

The top plot of Figure 3.6 shows the transmission in the first setup.

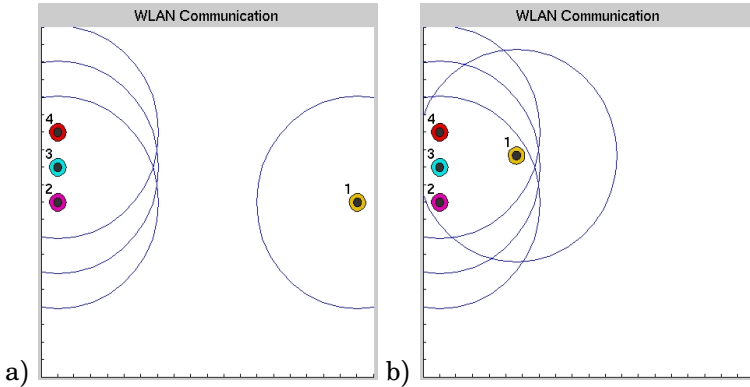


Figure 3.5 The two configurations considered in the first example in Section 3.7. Node 1, 2, and 3 all send to node 4. Node 1 is too far away to reach the destination in the first setup, whereas in the second setup all three sending nodes interfere. The circles around the nodes show the distance at which the transmitted signal is no longer possible to detect (receiver signal threshold).

At time 0.02 all three nodes start to send at the same time, and at the time when they have finished sending + $ACKtimeout$ they conclude that for some reason their messages were not received properly. All nodes then choose a randomized back-off time (according to the CSMA/CA policy) and the second time, nodes 2 and 3 both manage to get their messages sent to the receiver without further collisions.

Node 1, however, is trying to send its message over and over again. This depends on the fact that it is too far away from the intended receiver node and therefore does not get an ACK message back. So it sends its message, waits the $ACKtimeout$ and then tries to send it again after waiting an additional back-off time. This behavior continues until the maximum retry limit, which is set to five in this example, has been reached and then it drops the message.

In the second setup, the transmissions of all three sending nodes collide in their first try (as seen in the bottom plot of Figure 3.6). They all choose their random back-off times before re-transmitting again and getting their messages through in the second try.

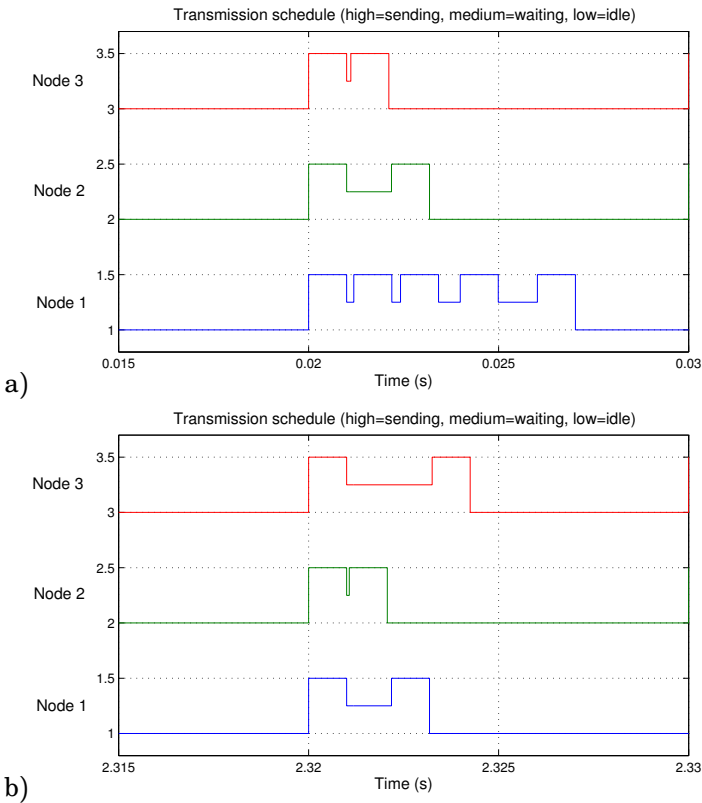


Figure 3.6 Close-ups of the network schedules corresponding to the two configurations described in Figure 3.5.

RoboCup

This setup is inspired by the well-known RoboCup project [The RoboCup Federation, 2004]. This project aims at, by the year of 2050, having a team of fully autonomous robots win against the human world soccer champions. RoboCup 2006 will be held in Bremen, Germany. This soccer cup consists of five leagues: the simulation league, the small size league, the middle size league, the 4-legged league, and the humanoid league.

This example involves two teams consisting of five players each, playing soccer against each other on a simulated play field. All the players are modeled as TrueTime computer blocks and the communication between them is performed via the wireless network block. The example shows that TrueTime is a good tool for evaluating high-level strategies in the field of distributed intelligence, autonomous agents and multi-agent collaboration.

Robot Modeling Each soccer playing robot, robot, is modeled by a TrueTime computer and simple dynamics for its planar motion. The Simulink subsystem representing a robot is shown in Figure 3.7. In this simplified model it is assumed that the robot moves independently in the x- and y-directions. The robot has two D/A outputs representing the currents to the motors. The dynamics between motor current and position is given by the transfer function $\frac{1}{s(s+3.5)}$, with $\frac{1}{s+3.5}$ being the transfer function between motor current and velocity. By integrating the velocity we obtain the position. These positions, i.e., the x and y coordinates, are then fed into the TrueTime wireless network block in order to calculate the path-loss of the radio signal.

It is further assumed that both the positions and the corresponding velocities are directly measurable, and thus the robot has four A/D inputs for these signals. In a more realistic setup, filtering of the position measurements would have been required to obtain velocity estimates.

The Schedule output is used to monitor the execution within the computer, and the robot communicates through the TrueTime wireless network using the Snd and Rcv ports.

Visualization and High-level Coordination The Simulink model of the soccer game consists of ten robots (two teams of five robots each) modeled according to the previous section. The x- and y-coordinates of each robot are also fed into a MATLAB S-function responsible for animating the game using 2D-graphics.

Within this framework, it is easy to evaluate different high-level strategies to coordinate the robots in the two teams. Both centralized and distributed intelligence may be considered. In the following we assume a centralized strategy where a master node is coordinating the movements of the robots in each team. The master has full sensing capability of both the ball position and the position of the individual

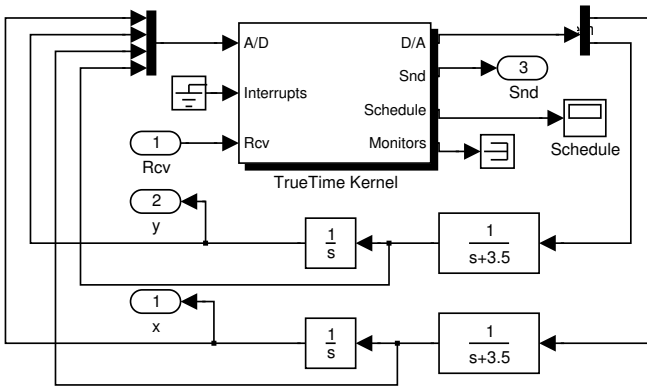


Figure 3.7 A model of a mobile robot in TrueTime, consisting of a TrueTime computer for code execution and communication and continuous-time dynamics for its motion.

robots. In a real setup this would, e.g., correspond to a camera monitoring the playing field and sending the information to the master node. The master node sends commands to robots in the team and receives in return responses of completed tasks. Commands can be, e.g., shoot, pass, dribble etc.

Local Sensing and Control The local intelligence of the individual robots can be chosen to incorporate different hardware configurations and physical devices. It is, e.g., straight-forward to model sensors such as proximity sensors and on-board vision systems.

The controller may also be arbitrarily advanced. The implementation in this case is intended to mimic a tiny embedded device without any RTOS support. The position controllers of the robots are implemented in interrupt handlers connected to hardware timers. Simple proportional control is used to follow velocity trajectories to reach reference positions provided by the master node. When a robot has the ball it either dribbles in the direction of the opposing goal or shoots if it is sufficiently close. When dribbling, the robot may also receive commands from the master to pass the ball in the direction of a team mate. Figure 3.8 shows screen shots from an attacking combination.

3.7 Simulation Case Studies

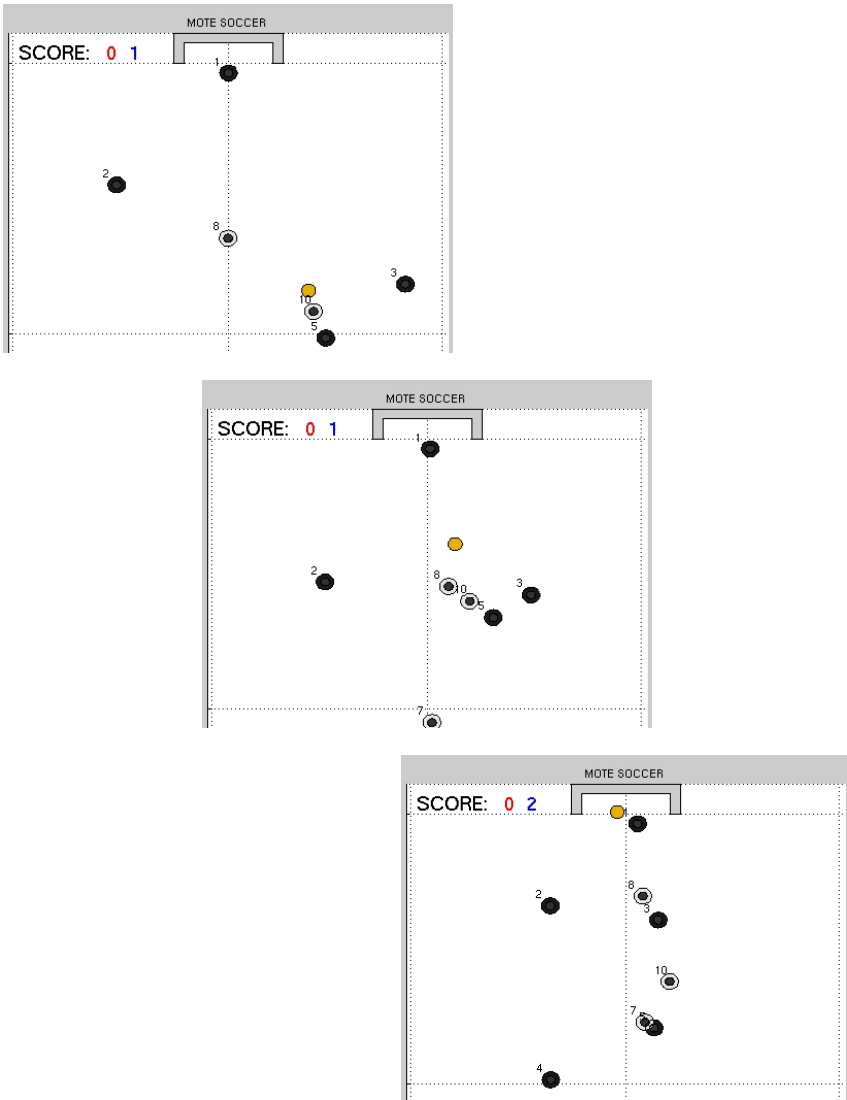


Figure 3.8 A sequence of screen shots from a successful attacking combination in the simulated RoboCup application.

3.8 Related Work

There exists a large number of general network simulators today. One of the most well-known is ns-2 [The VINT Project, 2004], which is a discrete-event simulator for both wired and wireless networks with support for, e.g., TCP, UDP, routing, and multi-cast protocols. It also supports simple movement models for mobile applications. The channel model in ns-2 is quite simple [Dricot and Doncker, 2004]. ns-2 makes the assumption that messages are received without errors if the power level is above a certain threshold. Packets with power levels below the same threshold are simply dropped. The packet with the largest power level is received if two transmissions occur at the same time, and the difference in power level between them is larger than 10 dB. Otherwise both packets are dropped. Three different path-loss models are available, two of them are deterministic and form ideal circles where the messages are received perfectly inside and dropped outside. The third model is called the shadowing model and adds some probabilistic changes to the path-loss by using a zero mean Gaussian variable. Another discrete-event computer network simulator is OMNeT++ [OMNeT++ Community, 2004]. It contains detailed IP, TCP, and FDDI protocol models and several other simulation models (file system simulator, Ethernet, framework for simulation of mobility, etc.). It uses the same path-loss function as the TrueTime wireless block, errors are however treated in a more detailed manner. It distinguishes between header and data part of packages and also between different modulation techniques. Compared to these simulators, the network simulation part in TrueTime is in some cases more simplistic. However, the strength of TrueTime is the co-simulation facilities that make it possible to simulate the latency-related aspects of the network communication in combination with the node computations and the dynamics of the physical environment. Rather than basing the co-simulation tool on a general network simulator and then try to extend this with additional co-simulation facilities, the approach has been to base the co-simulation tool on a powerful simulator for general dynamical systems, i.e., Simulink, and then add support for simulation of real-time kernels and the latency aspects of network communication to this. An additional advantage of this approach is the possibility to make use of the wide range of toolboxes that are available for MATLAB/Simulink.

For example, support for virtual reality animation.

There are also some network simulators geared towards the sensor network domain. TOSSIM [Levis *et al.*, 2003] compiles directly from TinyOS code and scales very well. Its radio and interference model is however very simplistic, with either perfect transmissions or predefined error rates which can be changed at runtime. COOJA [Österlind, 2006] is similar to TOSSIM but for the Contiki OS instead. Network in a box (NAB) [NAB, 2004] is another simulator for large-scale sensor networks. Another example is J-Sim, a general compositional simulation environment that includes a generalized packet switched network model that may be used to simulate wireless LANs and sensor network [Tyan, 2002]. Again, these types of simulators generally lack the possibility to simulate continuous-time dynamics and to simulate the inner workings of the nodes at the thread and interrupt handler level, features that have been present in TrueTime since the early versions.

A few other tools have been developed that support co-simulation of real-time computing systems and control systems. RTSIM [Palopoli *et al.*, 2000] has a module that allows system dynamics to be simulated in parallel with scheduling algorithms. XILO [El-Khoury and Törngren, 2001] supports the simulation of system dynamics, CAN networks, and priority-preemptive scheduling. Ptolemy II is a general purpose multi-domain modeling and simulation environment that includes a continuous-time domain, and a simple RTOS domain. Recently it has been extended in the sensor network direction [Baldwin *et al.*, 2004]. In [Branicky *et al.*, 2003] a co-simulation environment based on ns-2 is presented. The ns-2 simulator has been extended with an ODE-solver for dynamical simulations of the controller units and the environment. However, this tool lacks support for real-time kernel simulation.

3.9 Conclusions

This chapter has presented a simulation environment for mobile wireless networked embedded systems. The tool is focused on co-simulation, where the computer architecture in the form of computer nodes and communication networks are simulated in parallel with continuous-time dynamics modeling the physical environment.

The chapter has described the modeling and implementation of the

Chapter 3. A Simulation Tool for Wireless Control

IEEE 802.11b/g and 802.15.4 standards for wireless communication within the TrueTime framework. Other new features are simulation of local clocks, and power consumption within the individual nodes.

Two examples have been presented: a simple communication scenario and a mobile robot soccer game.

4

Future Work

4.1 The Nice Controller

The nice control of the CPU-bandwidth can be enhanced in some obvious ways. The most important is probably to make it possible to control the CPU-bandwidth on a per group basis instead of individually for each task. This is certainly doable, but the exact details has to be worked out with quite some care. When introducing more features such as controlling groups of tasks, it is also needed to create a better interface to make it easier to configure the controller. To gain more acceptance and feedback from the Linux community, the controller should be integrated into the CKRM project as soon as that project has stabilized its code base. This will also automatically give access to better configuration utilities.

Another point that needs to be given more thought, is identification of areas that could benefit from a controller like the one presented in this thesis. One of the obvious examples is web servers. Many web servers have customers that are not equally important. If tasks that service the customers are given different shares of the CPU-bandwidth, then they will also get different response times. This can be used to control the response times of the server. In that setup, the CPU-bandwidth controller will be used in a cascade structure.

4.2 The TrueTime Simulator

The programming of tasks in the TrueTime simulator would benefit if the “segment” structure could be left behind. Getting rid of the segment structure would make the code more similar to ordinary programming code, which would in turn increase the portability to and from other systems. As it is now, a context switch can only occur between two segments. It is of course possible to make a segment around every command if one wants to make the context switching more fine grained. That is, however, quite tedious work and it would be better if it could be done automatically.

Automatic code generation, both to and from TrueTime is another desired feature. With that available, it would be possible to use TrueTime for rapid prototyping of real-time control systems and then convert the code to a more efficient platform when properly analyzed. It would also be possible to analyze already available code to find bugs and performance bottlenecks. In these kinds of tests, it would be advantageous if the execution times used in TrueTime were based on values from real systems.

One problem with the current implementation, of both the wired and the wireless network blocks, is that they only remember the time of the next event. A better solution would be to calculate the time of all known future events and save them in a data structure. This data structure could then be updated when new events arrive.

The structure for implementing new network protocols could be enhanced and made pluggable, so that users will be able to implement and test protocols in a simpler manner. At the time of this writing, implementing a new network protocol in TrueTime requires too much knowledge about the inner details of the different blocks.

The network blocks only implement the MAC layers of the network. It would be advantageous from the users’ perspective if a common library containing higher level protocols could be implemented, such as TCP/IP or different routing protocols.

5

Bibliography

- Abeni, L. and G. C. Buttazzo (1998): “Integrating Multimedia Applications in Hard Real-Time Systems.” In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 4–13. Madrid, Spain.
- Abeni, L. and G. Lipari (2002): “Implementing Resource Reservations in Linux.” In *Proceedings of the Fourth Real-Time Linux Workshop*. Boston (MA).
- Abeni, L., G. Lipari, and G. C. Buttazzo (1999): “Constant Bandwidth vs Proportional Share Resource Allocation.” In *ICMCS '99: Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pp. 107–111. IEEE Computer Society, Washington, DC, USA.
- Andersson, M. (2006): “Strange Interactivity Behaviour.” Home page: <http://lkml.org/lkml/2006/2/27/104>.
- Andersson, M., D. Henriksson, and A. Cervin (2005): *TrueTime 1.3—Reference Manual*. Department of Automatic Control, Lund University, Sweden.
- Baldwin, P., S. Kohli, E. A. Lee, X. Liu, and Y. Zhao (2004): “Modeling of Sensor Nets in Ptolemy II.” In *IPSN'04: Proceedings of the Third International Symposium on Information Processing in Sensor Networks*, pp. 359–368. ACM Press.
- Branicky, M., V. Liberatore, and S. M. Phillips (2003): “Networked Control Systems Co-Simulation for Co-Design.” In *Proceedings of*

Chapter 5. Bibliography

- the 2003 American Control Conference*, vol. 4, pp. 3341–3346. Denver, USA.
- Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003): “How Does Control Timing Affect Performance?” *IEEE Control Systems Magazine*, **23:3**, pp. 16–30.
- CKRM (2006): “Class-based Kernel Resource Management (CKRM).” Home page: <http://ckrm.sourceforge.net/>.
- de Jongh, J. (2002): *Share Scheduling in Distributed Systems*. PhD thesis, Delft University of Technology.
- Dricot, J.-M. and P. D. Doncker (2004): “High-accuracy physical layer model for wireless network simulations in NS-2.” In *Proceedings of the International Workshop on Wireless Ad-hoc Networks, IWWAN'04*. Oulu, Finland.
- El-Khoury, J. and M. Törngren (2001): “Towards a Toolset for Architectural Design of Distributed Real-Time Control Systems.” In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. London, England.
- Epema, D. H. J. (1998): “Decay-Usage Scheduling in Multiprocessors.” *ACM Trans. Comput. Syst.*, **16:4**, pp. 367–415.
- Essick, R. B. (1990): “An Event-Based Fair Share Scheduler.” In *Proceedings of the Winter 1990 USENIX Conference*, pp. 147–162. USENIX.
- Fong, L. L. and M. S. Squillante (1995): “Time-Function Scheduling: A General Approach to Controllable Resource Management.” Technical Report RC 20155 (89194). IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598.
- Hellerstein, J. L. (1993): “Achieving Service Rate Objectives with Decay Usage Scheduling.” *IEEE Transactions on Software Engineering*, **19:8**, pp. 813–825.
- Hellerstein, J. L. (2004): “Challenges in Control Engineering of Computing Systems.” In *Proceedings of the 2004 American Control Conference*, vol. 3, pp. 1970–1979.

- Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2005): “Control Engineering for Computing Systems.” *IEEE Control Systems Magazine*, **25:6**, pp. 56–68.
- Henry, G. J. (1984): “The Fair Share Scheduler.” *AT&T Bell Laboratories Technical Journal*, **63:8**, pp. 1845–1857.
- IEEE (1999a): “ANSI/IEEE Std 802.11.”
- IEEE (1999b): “IEEE Std 802.11b.”
- Kay, J. and P. Lauder (1988): “A fair share scheduler.” *Communications of the ACM*, **31:1**, pp. 44–55.
- Levis, P., N. Lee, M. Welsh, and D. Culler (2003): “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications.” In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pp. 126–137. Los Angeles, CA, USA.
- Magnusson, P. S., M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner (2002): “Simics: A Full System Simulation Platform.” *IEEE Computer*, **35:2**, pp. 50–58.
- Moruzzi, C. and G. Rose (1991): “Watson Share Scheduler.” In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA '91)*, pp. 129–133. USENIX, San Diego, USA.
- NAB (2004): “NAB (Network in A Box).” Home page: <http://nab.epfl.ch/>.
- Nagar, S., R. V. Riel, H. Franke, C. Seetharaman, V. Kashyap, and H. Zheng (2004): “Improving Linux resource control using CKRM.” In *Proceedings of the 2004 Linux Symposium*, vol. 2, pp. 511–524. Ottawa, Ontario, Canada.
- OMNeT++ Community (2004): “OMNeT++ Discrete Event Simulation System.” Home page: <http://www.omnetpp.org>.
- Palopoli, L., L. Abeni, and G. Buttazzo (2000): “Real-time control system analysis: An integrated approach.” In *Proceedings of the 21st IEEE Real-Time Systems Symposium*. Orlando, Florida.

Chapter 5. Bibliography

- Rajkumar, R., K. Juvva, A. Molano, and S. Oikawa (1998): “Resource kernels: a resource-centric approach to real-time and multimedia systems.” In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98)*, pp. 150–164. San Jose, CA, USA.
- Schiller, J. (2003): *Mobile Communications Second Edition*. Addison-Wesley. ISBN 0 321 12381 6.
- Sprunt, B., L. Sha, and J. Lehoczky (1989): “Aperiodic Task Scheduling for Hard Real-Time Systems.” *Real-Time Systems Journal*, **1:1**, pp. 27–60.
- Stoica, I. and H. Abdel-Wahab (1995): “Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation.” Technical Report. Norfolk, VA, USA.
- The Mathworks (2001): *Simulink: A Program for Simulating Dynamic Systems – User’s Guide*. The MathWorks Inc., Natick, MA.
- The RoboCup Federation (2004): Home page: <http://www.robocup.org>.
- The VINT Project (2004): “The Network Simulator ns-2.” Home page: <http://www.isi.edu/nsnam/ns/index.html>.
- Tyan, H.-Y. (2002): *Design, realization and evaluation of a component-based compositional software architecture for network simulation*. PhD thesis, Ohio State University.
- Waldspurger, C. A. and W. E. Wehl (1995a): “Lottery Scheduling: Flexible Proportional-Share Resource Management.” In *First Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–11. USENIX Association.
- Waldspurger, C. A. and W. E. Wehl (1995b): “Stride Scheduling: Deterministic Proportional-Share Resource Management.” Technical Report MIT/LCS/TM-528. Massachusetts Institute of Technology, MIT Laboratory for Computer Science.
- Österlind, F. (2006): “A Sensor Network Simulator for the Contiki OS.” Technical Report T2006-05. SICS – Swedish Institute of Computer Science.

| | | | |
|---|------------------------------|--|--|
| Department of Automatic Control Lund University Box 118 SE-221 00 Lund Sweden | | <i>Document name</i> LICENTATE THESIS | |
| | | <i>Date of issue</i> August 2006 | |
| | | <i>Document Number</i> ISRN LUTFD2/TFRT--3240--SE | |
| <i>Author(s)</i> Martin Ohlin | | <i>Supervisor</i> Karl-Erik Årzén Anton Cervin Johan Eker | |
| | | <i>Sponsoring organisation</i> RUNES | |
| <i>Title and subtitle</i> Feedback Linux Scheduling and a Simulation Tool for Wireless Control | | | |
| <i>Abstract</i> <p>Computing systems are becoming more and more complex and powerful. It is nowadays not uncommon to run several server applications on the same physical platform. This gives rise to a need for resource reservation techniques, so that administrators may prioritize some tasks, or customers, over others. This thesis gives an introduction to the Linux kernel 2.6 task scheduler, and scheduling related operating system concepts such as priority, nice value, interactivity and task states. The thesis also presents an implementation of a scheduling mechanism, that in a non-intrusive way introduces per task CPU bandwidth reservations in the Linux operating system.</p> <p>The MATLAB/Simulink-based simulator TrueTime is given a short introduction, and the wireless capabilities of the tool are described in more detail. TrueTime is a tool for co-simulation of real-time tasks, network communication, and continuous-time plant dynamics. The modeling of the common medium access control (MAC) layers of IEEE 802.11 and IEEE 802.15.4 is described, along with the radio model used. TrueTime's capabilities to simulate local clocks with drift, Dynamic Voltage Scaling, and battery powered devices are also presented.</p> | | | |
| <i>Key words</i> Linux, Scheduling, Feedback Scheduling, Resource Reservation, Simulation Tools | | | |
| <i>Classification system and/ or index terms (if any)</i> | | | |
| <i>Supplementary bibliographical information</i> | | | |
| <i>ISSN and key title</i> 0280-5316 | | <i>ISBN</i> | |
| <i>Language</i> English | <i>Number of pages</i> 78 | <i>Recipient's notes</i> | |
| <i>Security classification</i> | | | |

